Python GTK+ 3 Tutorial Documentation

Andrew Steele

Oct 29, 2018

CONTENTS

Author: Andrew Steele
Last updated: Oct 26, 2018

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Before using this tutorial

Prior to working through this tutorial, it is recommended that you have a reasonable grasp of the Python programming language. GUI programming introduces new challenges compared to interacting with the command line. It is necessary for you to know how to create and run Python files, understand basic interpreter errors and warnings, work with strings, integers, floats, and Booleans. For the advanced widgets covered in this tutorial, good knowledge of lists and tuples will also be required.

Prior knowledge of GTK+ is not required.

1.2 About this tutorial

This guide assumes that you are using the Python 3.4 series and that you are using GTK+ 3.16 and an as-up-to-date version of Python-GObject as possible. The newer the Python-GObject version, the less likely there are to be issues. Older versions will work for some examples.

To see which version of GTK+ you have installed, run the following script to check the verison number.

Download: `Version`

1.3 Deprecation

Deprecation is the process of preparing features within software to be decommisioned and removed. GTK+ deprecates objects by providing warnings both in the GTK+ framework and documentation prior to removal at a later date. This allows developers to change their code before a feature is removed. In many cases however, a feature is not removed until the next major release to prevent mass breakage of applications.

Reasons for deprecation generally include a feature being superseded, or a feature no longer being widely used.

It is highly recommended to not use deprecated features of GTK+ or Python when developing applications, particularly when learning as they can cause problems when understanding other areas of development.

This tutorial does not cover widgets which have been marked as deprecated. Any widgets marked as deprecated by GTK+ in the future will also be removed from future versions of the tutorial.

1.4 Credits

The following people/teams deserve credit and provided me with the ability to write this tutorial:

- Guido van Rossum and other developers for creating and maintaining the Python programming language.
- The GTK+ developers for their excellent work on creating the toolkit.
- James Henstridge and other developers for their work on the PyGTK bindings.
- John Finlay, Rafaele Villar Burke and other maintainers of the original (and excellent) PyGTK tutorial.
- The Introspection binding authors.

1.5 Contact

Please contact me at andrew@andrewsteele.me.uk to report issues, bugs and provide comments. All suggestions are welcome.

1.6 License

- The tutorial text and code examples are released under a CC0 1.0 Universal (CC0 1.0) license (this essentially makes them Public Domain).
- The GTK+ logo used for examples and the site favicon is released under the Creative Commons Attribution-Share Alike 3.0 Unported license.

CHAPTER

TWO

GETTING STARTED

To begin, the below is a simple example which creates a 200 pixel width by 200 pixel high window. You will not be able to close the window in the traditional way by clicking the X in the corner. Instead you will need to exit the process via kill command.

```
#!/usr/bin/env python3

from gi.repository import Gtk

class GettingStarted:
    def __init__(self):
        window = Gtk.Window()
        window.connect("destroy", Gtk.main_quit)
        window.show()

GettingStarted()
Gtk.main()
```

Download: `Getting Started`

To run the application, open a terminal window and enter:

```
python gettingstarted.py
```

Python files can be run as with a standard application by double-clicking on the file once it has been made executable.

Due to this example having no way to quit via the GUI, we need to use `CONTROL+4` to exit. If you click on the X (close) button, you will notice that the terminal does not return to the prompt. The application at this point is still running and consuming CPU/RAM. Running applications from the terminal while developing can allow you to spot issues such as this (as well as see warnings/errors that may appear).

We will look at signals shortly and show how they can be connected to ensure the application works correctly.

2.1 Hello World

As with any programming tutorial, there needs to be a 'Hello World' example. We will build on the previous example to create an application that performs a function and produces an output.

```
#!/usr/bin/env python3

from gi.repository import Gtk

class HelloWorld(Gtk.Window):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

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

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

window = HelloWorld()
window.show_all()

Gtk.main()
```

Download: `Hello World`

Once downloaded, open a terminal and run the application as with the previous example.

2.2 Comments

Comments in Python applications can be specified in two ways:

- Using a `#` to designate a single-line comment.
- Using `'''` to specify a block comment. The `'''` is used both at the beginning and end of the text to specify all text within becomes a comment.

In a number of text editors, commented code will turn a different colour.

It is highly recommended to use comments both when learning to remind yourself of a particular piece of code, and when developing in the future to both guide yourself and others who may utilise your code.

CHAPTER

THREE

PACKING THEORY

Our first 'Hello World' program only contains a single widget. However, in the real word we will require more than one widget per Window. For this purpose, we can use several different widgets to achieve the layout we want. The most commonly used are Box and Grid widgets. These are known as packing, or container widgets, and are invisible to the user. They are placed within a Window and each other to develop a complex interfaces.

For anybody who has used Microsoft Visual Studio, placing widgets in GTK+ (and virtually all other toolkits such as Qt, wx) is different. Microsoft products use a pixel-based system where the widget is dropped in place and adjusted to size as needed. GTK+ uses a packing-based method where widgets are placed in container widgets, and adjusted to the size they are allocated.

The learning curve of the packing-based method is arguably slightly higher, however it allows for rapid development and fluid interface design once the initial learning curve is overcome. People who have worked with HTML and CSS will find the GTK+ packing behaviour similar.

Due to the fluid nature of the packing system, when a container (such as a Window or Grid) is resized, the child content within it is adjusted to fit correctly. If the size is increased, the child content becomes bigger, regardless of what else is contained within. If it is shrunk, the child content becomes as small as possible.

Most widgets have a minimum size which is dependent on their content. For example, a Button can not be made smaller than its Label which means that attempting to make the Window surrounding the Button smaller than the Label within will fail.

3.1 Height-for-width

GTK+ uses a height-for-width system to manage the size to widgets. In the most basic term, the widget can adjust how much vertical space it requires based on the horizontal space it is allocated.

For example, a multi-line text label will shrink its parent widget as it requires fewer lines due to increased width.

CHAPTER

FOUR

SIGNAL THEORY

GTK+ responds to events which occur while the application is waiting in `gtk.main()`. When an event, such as a mouse click occurs, control is passed to the appropriate function before returning to `gtk.main()`.

Control is passed using signals. When an event occurs, the appropriate signal is emitted by the widget that was interacted with. The type of event used by a widget depends on what it is. For example a Button has a `"clicked"` event, however a Label does not.

A generic example of a signal is:

```
widget.connect("event", function, data)
```

Firstly, the widget is the name of the widget which have created at construction time. There are many widgets which use signals to communicate with other areas of the application, for example a Button. Next, the event method is the action which has been performed. Each widget has its own particular events which can be emitted. Using our Button example, we would usually set the event to `"clicked"`. This means that when the Button is clicked, the signal is emitted. Finally the data argument should be passed when the signal is issued and provides the function with extra parameters it may require.

When the `widget.connect()` code is run, a *handler_id* is returning. This is a unique number identifying that particular signal throughout the runtime of the application. To retrieve the handler_id when connecting use:

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

To disconnect this at a later date:

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

4.1 Arguments Error

When connecting signals from widgets, it is important to pass the correct number of parameters. The number of parameters, and the type depend on the signal being connected. If the number of parameters passed is greater than the number of parameters a function can receive, an error similar to below will be displayed:

```
TypeError: function() takes exactly 2 arguments (3 given)
```

Alternatively, if the number of parameters being provided is fewer than the number that a function can receive, an error similar to below will be displayed:

```
TypeError: function() takes exactly 4 arguments (1 given)
```

CHAPTER

FIVE

HIERARCHY THEORY

Widgets in GTK+ are built on top of each other to create what the user views on screen. This is done to varying degrees of complexity depending on the widget, with the lines blurred between some and more clear between others.

One of the useful features of widget hierarchy is that it allows code to be shared between widgets, so that they behave in similar ways making development much easier once the initial learning curve is overcome.

5.1 Example

All visible widgets are based on GtkWidget. This is a base class which contains a range of methods. These methods can be used by any widget which inherits from GtkWidget. One example would be the `.set_tooltip_text()` method which allows a string of text describing the widget to be attached to it.

This means that the method, due to inheritance, can also be used by GtkButton or GtkEntry.

5.2 Structure

The structure of widgets and objects in GTK+ is shown below:

```
+GtkWidget
+---GtkContainer
+------GtkBin
+---------GtkWindow
+------------GtkDialog
+---------------GtkAboutDialog
+---------------GtkAppChooserDialog
+---------------GtkColorSelectionDialog
+---------------GtkFileChooserDialog
+---------------GtkFontChooserDialog
+---------------GtkMessageDialog
+---------------GtkPageSetupUnixDialog
+---------------GtkPrintUnixDialog
+---------------GtkRecentChooserDialog
+------------GtkAssistant
+------------GtkOffscreenWindow
+------------GtkPlug
+---------GtkAlignment
+---------GtkComboBox
+------------GtkAppChooserButton
+------------GtkComboBoxText
+---------GtkFrame
```

(continues on next page)

(continued from previous page)

```
+------------GtkAspectFrame
+---------GtkButton
+------------GtkToggleButton
+---------------GtkCheckButton
+-------------------GtkRadioButton
+------------GtkColorButton
+------------GtkFontButton
+------------GtkLinkButton
+------------GtkLockButton
+------------GtkScaleButton
+---------------GtkVolumeButton
+---------GtkMenuItem
+---------GtkEventBox
+---------GtkExpander
+---------GtkHandleBox
+---------GtkToolItem
+---------GtkOverlay
+---------GtkScrolledWindow
+---------GtkViewport
+------GtkBox
+------GtkFixed
+------GtkGrid
+------GtkPaned
+------GtkIconView
+------GtkLayout
+------GtkMenuShell
+------GtkNotebook
+------GtkSocket
+------GtkTable
+------GtkTextView
+------GtkToolbar
+------GtkToolItemGroup
+------GtkToolPalette
+------GtkTreeView
```

CHAPTER

SIX

GTK+ MAIN LOOP

GTK+ is event-driven by nature in that actions or procedures are performed when an event occurs within the application. Take for example the 'Hello World' application from earlier:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class HelloWorld:
    def close_hello_world(self, widget):
        Gtk.main_quit()

    def print_hello_world(self, widget):
        print("Hello World")

    def __init__(self):
        window = Gtk.Window()
        window.connect("destroy", self.close_hello_world)

        button = Gtk.Button("Click here")
        button.connect("clicked", self.print_hello_world)
        window.add(button)

        window.show_all()

HelloWorld()
Gtk.main()
```

Download: `GTK+ Main Loop 1`

After the `HelloWorld()` class has run, the application loops in the `Gtk.main()` line of code. This loop waits for an event in the application to occur such as a click, drag-and-drop, or any other number of events.

A Python application can either work as standalone or part of another application. However, if the application is being used as part of another, the GTK+ Main Loop position will cause a problem as it stands. Therefore, we only need to allow the Main Loop to run if the application is running standalone. We will need to change the code to the following:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class HelloWorld:
    def close_hello_world(self, widget):
        Gtk.main_quit()
```

(continues on next page)

(continued from previous page)

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

    def __init__(self):
        window = Gtk.Window()
        window.connect("destroy", self.close_hello_world)

        button = Gtk.Button("Click here")
        button.connect("clicked", self.print_hello_world)
        window.add(button)

        window.show_all()

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

Download: `GTK+ Main Loop 2`

CHAPTER

SEVEN

WINDOW

The Window is the basis of most applications created using GTK+. It is the widget which provides the framework on which other widgets can be added; therefore it is known as a container widget.

On its own, a Window can only pack a single widget within its container.

7.1 Constructor

The Window can be constructed using:

```
window = Gtk.Window()
```

7.2 Methods

Widgets can be added to or removed from the Window with the following:

```
window.add(widget)
window.remove(widget)
```

A title can also be displayed on the Window with:

```
window.set_title(text)
```

By default, the size of the Window is 200x200 pixels when there is no content within the Window. This can be changed at launch with:

```
window.set_default_size(width, height)
```

In some cases, it may only be required to set the width or the height. To set a size dynamically, use -1. This will then size the Window to that of the child content.

To maximize, unmaximize, minimize or unminimize a Window, the following methods can be used:

```
window.maximize()
window.unmaximize()
window.iconify()
window.deiconify()
```

In some cases, it is useful to allow the Window to be made fullscreen. The methods used to make a Window fullscreen or unfullscreened are:

```
window.fullscreen()
window.unfullscreen()
```

By default, all Windows allow the user to resize them as necessary. This automatically adjusts the content and re-flows it as per the Window size. The ability to allow resizes can be configured with:

```
window.set_resizable(resizable)
```

When *resizable* is set to False, two changes are made to the Window. First, there will be no maximize button on the title bar. Second, the user can now adjust the Window size by grabbing the border of the Window.

A resize grip is positioned in the bottom-right corner of all windows which are resizable. To disable this run:

```
window.set_has_resize_grip(resize_grip)
```

When *resize_grip* is set to `False`, the resize grip will be removed.

To make your application easily identifiable, you can specify an Window icon to be set with:

```
window.set_icon_from_file(file_path)
```

The *file_path* parameter should be set to that of an image such as a PNG, SVG, JPG, GIF, XPM or BMP (others are supported but those are the most common).

To set the focus of a particular widget at program execution time, use:

```
window.set_focus(child)
```

In some cases, it may be necessary to display a Window on top of another Window, and prevent interaction with the Window on the bottom. This can be achieved by setting the top Window as modal with:

```
window.set_modal(modal)
```

When *modal* is set to True, the bottom Window can not be interacted with by the user until the modal Window has been closed.

Another useful feature when working with multiple Window widgets is to make any child Windows related to their parent. This allows for neatness when the Windows are positioned. This can be done by:

```
window.set_transient_for(child)
```

The *child* should be the name of another Window.

When setting a Window to be transient for another, it is also useful to allow the child Window to be destroyed if the parent is with:

```
window.set_destroy_with_parent(destroy)
```

By default, all Window widgets created are decorated with a title bar that often contains icons, the application title and buttons. To control whether the Window Manager decorates the Window, use:

```
window.set_decorated(decorated)
```

When *decorated* is set to False, GTK+ will ask the Window to remove the Window decorations. Depending on the Window Manager in use and the platform being used, this may not have any effect.

Custom title bars can be set on the Window with the following method:

```
window.set_titlebar(child)
```

The *child* is a widget, most likely a *Grid* or *Box* with other child widgets. When this option is in use, GTK+ will tell the window manager not to draw a standard titlebar on the Window.

Note: GTK+ will tell the window manager not to display a titlebar, however the window manager is free to ignore this function if it is not supported.

7.3 Signals

The commonly use signals of a Window are:

```
"destroy"
"delete-event"
"event"
"add"
"remove"
```

The "destroy" signal is emitted when the user requests that the Window be destroyed. This is often emitted from the X button on the Title Bar of the Window. A "delete-event" is much the same as "destroy", however it is a request rather than a command. This is useful when wanting to confirm a close rather than command it. An "event" signal is used to notify on any events relating to the Window. The "add" and "remove" signals emit when child widgets are added to or removed from the Window.

7.4 Example

Below is an example of a Window:

```
#/usr/bin/env python3

from gi.repository import Gtk

class Window(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title("Window")
        self.connect("destroy", Gtk.main_quit)

window = Window()
window.show()

Gtk.main()
```

Download: `Window`

CHAPTER

EIGHT

BOX

A Box object is an invisible container which allows packing of child widgets in two modes; vertically-packed and horizontally-packed.

For more information on packing and the theory behind it using GTK+, see the *Packing Theory* page.

..note

```
The :doc:`grid` widget is recommended in many cases where a Box would have␣
↪traditionally been used.
```

8.1 Constructor

The Box can be constructed using the following:

```
box = Gtk.Box(orientation, spacing)
```

The *orientation* parameter should be set to one of the following values; `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`. By default, the Box is constructed with the `Gtk.Orientation.HORIZONTAL` orientation. The *spacing* value must be an integer value which indicates the amount of pixels between each of the child widgets.

8.2 Methods

Items can be packed using two methods; packing at the start of the container or packing at the end. When using a vertical Box, items using `.pack_start()` are packed from to the top. If using a horizontal box, packing using `.pack_start()`, child widgets are added to the left. This is done with:

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

The *child* parameter should be the name of the child widget that is being added to the Box. The *expand* property when set to `True` indicates that the child should be given extra space if the Box has room for it. When *fill* is set to `True`, the child widget is allocated the full horizontal or vertical space. The *padding* parameter should be set to an integer value which indicates how much space is put between it and other child widgets in the Box.

Items can also be removed from the Box with:

```
box.remove(child)
```

To reorder child widgets based on position use:

```
box.reorder_child(child, position)
```

The *position* value should be an integer, with `0` indicating the first position within the container. Alternatively, negative numbers can be entered which indicates a position from the end of the container.

The orientation of the Box can be changed after construction by:

```
box.set_orientation(orientation)
```

Again, the *orientation* value must be set to either `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`.

Child widgets can be reordered within the Box using:

```
box.reorder_child(child, position)
```

The *child* value should be the name of the widget which is to be moved. The *position* value should be an integer value which designates the new position of the child. `0` designates the first position in the box.

To ensure that all child widgets are set to an equal size regardless of their content, use:

```
box.set_homogeneous(homogeneous)
```

By default, *homogeneous* is set to `False` and all child widgets are sized based on their content.

The Box spacing can be set after construction with:

```
box.set_spacing(spacing)
```

8.3 Example

Below is an example of a Box:

```
#/usr/bin/env python3

from gi.repository import Gtk

class Box(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        hbox = Gtk.Box()
        hbox.set_orientation(Gtk.Orientation.HORIZONTAL)
        hbox.set_spacing(5)
        self.add(hbox)

        label = Gtk.Label(label="Label 1")
        hbox.pack_start(label, True, True, 0)
        label = Gtk.Label(label="Label 2")
        hbox.pack_start(label, True, True, 0)

        vbox = Gtk.Box()
        vbox.set_orientation(Gtk.Orientation.VERTICAL)
        vbox.set_spacing(5)
```

(continues on next page)

(continued from previous page)

```
        hbox.add(vbox)

        label = Gtk.Label(label="Label 3")
        vbox.pack_start(label, True, True, 0)
        label = Gtk.Label(label="Label 4")
        vbox.pack_start(label, True, True, 0)

window = Box()
window.show_all()

Gtk.main()
```

Download: `Box`

CHAPTER

NINE

GRID

A Grid widget is an invisible container which allows packing of one or more child widgets into rows and columns. It also supports height-for-width geometry means that widgets are scaled to sizes relative to the size of the space they are allocated.

For more information on packing and the theory behind it using GTK+, see the *Packing Theory* page.

9.1 Constructor

The Grid can be constructed using the following:

```
grid = Gtk.Grid()
```

9.2 Methods

To attach a child widget to the Grid, call:

```
grid.attach(child, left, top, width, height)
```

The *child* parameter should be the name of the widget which is to be placed within the Grid. The *left* and *top* values should be the row and column numbers which specify where the widget will be placed and the *width* and *height* parameters should be integer values indicating how many rows and columns the widget will span.

When adding multiple widgets to a Grid, it can be time-consuming to call `.attach()` repeatedly and enter the correct row and column values. To attach next to a previously added widget, use:

```
grid.attach_next_to(child, sibling, side, width, height)
```

The *child* again should be the name of the widget being added to the Grid. The *sibling* is the name of the widget the child is being added next to. The *position* should be one of `Gtk.PositionType.LEFT`, `Gtk.PositionType.RIGHT`, `Gtk.PositionType.TOP`, or `Gtk.PositionType.BOTTOM`. The *width* and *height* parameters should be integer values indicating how many rows and columns the widget will span.

To remove columns and rows from the Grid, use the method:

```
grid.remove_column(position)
grid.remove_row(position)
```

The *position* value is the number of the column or row to be removed. The use of these methods causes the following:

- Any widgets in the column or row are removed

- Any widgets which span the column or row have their height or width shortened
- Any widgets to the right or bottom are shifted

To set the spacing between rows and columns:

```
grid.set_row_spacing(spacing)
grid.set_column_spacing(spacing)
```

For neatness, it may be useful to set all widgets in a row or column to be an equal size with:

```
grid.set_row_homogeneous(homogeneous)
grid.set_column_homogeneous(homogeneous)
```

When *homogeneous* is set to True, the widgets will take the same size with the largest child widget dictating the size of all others in the row or column.

To retrieve a widget located at a particular position within the Grid call:

```
grid.get_child_at(left, top)
```

The *left* and *top* values should be specified as integer values identifying the location of the child.

9.3 Example

Below is an example of a Grid:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Grid(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        grid = Gtk.Grid()
        grid.set_row_spacing(5)
        grid.set_column_spacing(5)
        self.add(grid)

        button = Gtk.Button(label="Button 1")
        grid.attach(button, 0, 0, 1, 2)
        button = Gtk.Button(label="Button 2")
        grid.attach(button, 1, 0, 1, 1)
        button = Gtk.Button(label="Button 3")
        grid.attach(button, 2, 0, 1, 1)
        button = Gtk.Button(label="Button 4")
        grid.attach(button, 1, 1, 2, 1)

window = Grid()
window.show_all()

Gtk.main()
```

Download: `Grid`

CHAPTER

TEN

LABEL

A Label widget can be used to display anything from small to large amounts of text. The text can be formatted in a variety of different ways such as bold, italic or underline.

10.1 Constructor

The Label can be constructed using the following:

```
label = Gtk.Label(label)
```

The *label* parameter should be set to display the text within the Label widget.

10.2 Methods

To set the text of the Label after the construction of the Label, use either:

```
label.set_text(label)
label.set_label(label)
```

The *label* parameter should be a string. Any other values such as integers or floats should be converted before attempting to display on the Label. Text can also be set within the Label using 'n' and 't' formatting characters which provide new lines and tab spacing respecitively.

To retrieve the text set on the Label call:

```
label.get_text()
```

If the Label should use markup tags rather than just displaying them as text, this can be enabled with:

```
label.set_use_markup(use_markup)
```

The markup string, as opposed to plain text should be specified within the Label by calling:

```
label.set_markup(markup)
```

Markup allows the text in the Label to be customised, providing markup values similar to those used in HTML. Some examples include:

- Bold
- Italics

- <u>Underline</u>
- Link

By default, text within the Label can not be selected by the user. This can be changed with:

```
label.set_selectable(selectable)
```

When *selectable* is set to `True`, the user will be able to highlight the text within the Label.

When using multi-line text, the justification can be manipulated using:

```
label.set_justify(justify)
```

The default setting is `Gtk.Justification.LEFT`, however `Gtk.Justification.RIGHT`, `Gtk.Justification.CENTER` and `Gtk.Justification.FILL` can also be used.

If the Label widget is to wrap lines, it may be useful to set how many lines the Label wraps to:

```
label.set_lines(lines)
```

The *lines* value takes an integer value as the number of lines, or `-1` if the number of lines is not to be set.

Single line mode can also be enforced on the Label with the call:

```
label.set_single_line_mode(mode)
```

When *mode* is set to `True`, the text will not be split across multiple lines.

Text can be wrapped in the label if required with:

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

The *wrap* setting in `.set_line_wrap()` when set to `True` enforces line wrapping if the line is too long.

The text within the label can be aligned both horizontally and/or vertically with:

```
label.set_xalign(xalign)
label.set_yalign(yalign)
```

The *xalign* and *yalign* properties should be a value between `0.0` and `1.0`, with `0.0` indicating left or top and `1.0` indicating right or bottom.

The alignment can also be retrieved via:

```
label.get_xalign()
label.get_yalign()
```

By default, the label alignment values are 0.5 (centered) for both horizontal and vertical planes.

Text in a label can be angled, to orient it in a different direction, with the value supplied indicating an angle of orientation:

```
label.set_angle(angle)
```

The *angle* parameter must be a number between `0` and `360`.

Mnemonic keys provide access to widgets via a keyboard shortcut, with an underscore before the key to be used. This is tied to the widget using the method:

```
label.set_mnemonic_widget(widget)
```

The *widget* is another widget in the application associated with the label. Typically this is usually an *Entry* or *SpinButton*.

Label widgets typically expand to fit the content, however in some cases it may be suitable to set a target width and maximum width in characters with:

```
label.set_width_chars(width)
label.set_max_width_chars(max_width)
```

As a Label can contain links, it can be configured to remember whether a link has been accessed. This is displayed similar to how a browser does, with the link changing colour:

```
label.set_track_visited_links(track_links)
```

When *track_links* is set to `True`, the tracking will be enabled for that Label widget.

A URI can be obtained from the Label by calling:

```
label.get_current_uri()
```

The URI returned is often used in the `"active-link"` signal or when querying for a *Tooltip*.

10.3 Signals

The commonly used signals of a Label are:

```
"activate-link" (label, uri)
```

The *uri* parameter of the signal passes the link when the user clicks on the label.

10.4 Example

Below is an example of a Label:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Label(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(600, -1)
        self.connect("destroy", Gtk.main_quit)

        grid = Gtk.Grid()
        grid.set_border_width(5)
        grid.set_row_spacing(5)
        grid.set_column_spacing(5)
        self.add(grid)

        label = Gtk.Label("An example of a Label widget.")
```

(continues on next page)

(continued from previous page)

```
        label.set_selectable(True)
        grid.attach(label, 0, 0, 1, 1)
        label = Gtk.Label("This is a label spread across multiple\nlines using the␣
↪newspace character\nto indicate the line break.")
        grid.attach(label, 0, 1, 1, 1)
        label = Gtk.Label("Tab spaces\tcan also be\tdefined if required.")
        grid.attach(label, 0, 2, 1, 1)
        label = Gtk.Label("Label widgets can also accept underline patterns.")
        label.set_pattern("_____                              _________")
        label.set_line_wrap(True)
        grid.attach(label, 0, 3, 1, 1)

        label = Gtk.Label("Justification options are\nable to align text in the label,
↪\nsuch as to the left.")
        label.set_justify(Gtk.Justification.LEFT)
        grid.attach(label, 1, 0, 1, 1)
        label = Gtk.Label("Centering of text is possible\nto ensure that the␣
↪margin\nof each sentence is even.")
        label.set_justify(Gtk.Justification.CENTER)
        grid.attach(label, 1, 1, 1, 1)
        label = Gtk.Label("Text can also be right-justified\nto align to the right␣
↪hand\nmargin of the label.")
        label.set_justify(Gtk.Justification.RIGHT)
        grid.attach(label, 1, 2, 1, 1)
        label = Gtk.Label("Content is also justifiable to ensure that the sentences␣
↪are evenly distributed. This ensures that the endings of each lines match. It does␣
↪however require line wrapping to be enabled, and there are no manual breaks.")
        label.set_line_wrap(True)
        label.set_justify(Gtk.Justification.FILL)
        grid.attach(label, 1, 3, 1, 1)

        label = Gtk.Label("An angle can also be specified\nto orient the text.")
        label.set_angle(90)
        grid.attach(label, 2, 0, 1, 3)
        label = Gtk.Label("<a href='http://programmica.com/'>A website link</a>")
        label.set_use_markup(True)
        grid.attach(label, 2, 3, 1, 1)

label = Label()
label.show_all()

Gtk.main()
```

Download: `Label`

CHAPTER

ELEVEN

BUTTON

The Button widget is commonly used to allow a user to run a command or operation. It can display text and/or icons and provides an easy way for the user to interact with the application.

11.1 Constructor

The Button can be constructed using the following:

```
button = Gtk.Button(label)
```

The *label* parameter used in the first constructor allows the entering of text to display on the Button.

11.2 Methods

To set the text on the Button after construction:

```
button.set_label(text)
```

Images can also be set on Button widgets with:

```
button.set_image(image)
```

By default, all Button widgets have a border around them. This can be configured using:

```
button.set_relief(relief_style)
```

The *relief_style* parameter can be set as follows, with the default being `Gtk.BorderRelief.NORMAL`. The alternatives are `Gtk.ReliefStyle.HALF` or `Gtk.ReliefStyle.NONE`.

It is good practice to use a mnemonic in the label. This requires an underscore inserted into the label (e.g. "_Cancel"). GTK+ parses the underscore and converts it into an underline beneath the following character, which the user can then access as a shortcut to the function.

    button.set_use_underline(True)

When a Button is clicked, the focus is changed to that of the Button. To prevent this happening the following method can be used:

```
button.set_focus_on_click(focus)
```

If *focus* is set to `False`, the focus will be retained on whichever widget had it last before clicking the Button.

To force a button to show an image, even if the option is disabled globally, use:

```
button.set_always_show_image(show_image)
```

11.3 Signals

The commonly used signals of a Button are:

```
"clicked" (button)
"pressed" (button)
"released" (button)
```

The `"clicked"` signal is emitted when the user presses, then releases the mouse button when on the Button. A `"pressed"` signal emits when the mouse button is pressed above the Button while `"released"` emits when the mouse button is released over the Button. In most cases, the `"clicked"` signal should be used.

11.4 Example

Below is an example of a Button:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Button(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

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

    def on_button_clicked(self, button):
        print("Button has been clicked!")

window = Button()
window.show_all()

Gtk.main()
```

Download: `Button`

CHAPTER

TWELVE

TOGGLEBUTTON

A ToggleButton provides a way to indicate three modes of operation; active, inactive and inconsistent. A ToggleButton is used to indicate whether an option is enabled or disabled.

A ToggleButton is based on a *Button* meaning that they use many methods in the same way.

12.1 Constructor

A ToggleButton can be constructed using the following:

```
togglebutton = Gtk.ToggleButton(label)
```

12.2 Methods

To retrieve the state of a ToggleButton use:

```
togglebutton.get_active()
```

Setting the state of the ToggleButton programatically can be done with:

```
togglebutton.set_active(active)
```

When *active* is set to `True`, the ToggleButton will appear in a depressed state.

An inconsistent state can be set on a ToggleButton which can be used to indicate whether other widgets are at the correct values. For example, if three ToggleButtons are a mix of active and inactive, the fourth may display an inconsistent state. This can be retrieved with:

```
togglebutton.get_inconsistent()
```

Set the inconsistent parameter on the following method to `True` to activate the inconsistent state:

```
togglebutton.set_inconsistent(inconsistent)
```

12.3 Signals

The commonly used signals of an ToggleButton are:

```
"toggled" (togglebutton)
```

When the ToggleButton widget is clicked, the `"toggled"` signal is emitted. This occurs when the state is changed from active and inactive, and vice-versa.

12.4 Example

Below is an example of a ToggleButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def togglebutton_toggled(togglebutton):
    print("ToggleButton has been toggled!")

window = Gtk.Window()
window.connect("destroy", Gtk.main_quit)

togglebutton = Gtk.ToggleButton(label="ToggleButton")
togglebutton.connect("toggled", togglebutton_toggled)
window.add(togglebutton)

window.show_all()

Gtk.main()
```

Download: `ToggleButton`

CHAPTER

THIRTEEN

CHECKBUTTON

A CheckButton displays a small box which is allowed to be in one of three states; checked, unchecked or inconsistent. It is displayed with a Label next to it indicating what function the CheckButton performs.

A CheckButton is based on the *ToggleButton* widget, and inherits many of the same methods, properties and signals.

13.1 Constructor

The CheckButton can be constructed using:

```
checkbutton = Gtk.CheckButton(label)
```

The *label* parameter allows the associated text to be defined at construction time.

13.2 Methods

The label on the CheckButton is definable after construction via:

```
checkbutton.set_label(label)
```

It is good practice to use a mnemonic in the label. This requires an underscore inserted into the label (e.g. "_Cancel"). GTK+ parses the underscore and converts it into an underline beneath the following character, which the user can then access as a shortcut to the function.

checkbutton.set_use_underline(True)

By default, the CheckButton will be in the inactive (unchecked) state. To set the state the following can be used:

```
checkbutton.set_active(active)
```

The *active* parameter should be set to either `True` which sets the CheckButton to ticked, or `False` which is unticked.

To retrieve the state of the CheckButton:

```
checkbutton.get_active()
```

In some cases, the CheckButton may be set to an inconsistent state, which is used to indicate the status of other CheckButton widgets. For example, three CheckButton's may be a mix of checked and unchecked, which leaves the fourth set as inconsistent. This can be set programatically with:

```
checkbutton.set_inconsistent(inconsistent)
```

To retrieve whether a CheckButton is set as inconsistent use:

```
checkbutton.get_inconsistent()
```

If the CheckButton is in an inconsistent state, `True` will be returned.

13.3 Signals

The commonly used signals of a CheckButton are:

```
"toggled" (checkbutton)
```

A `"toggled"` signal emits from the CheckButton when the mode is changed to active or inactive.

13.4 Examples

Below is an example of a CheckButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class CheckButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        checkbutton = Gtk.CheckButton(label="CheckButton")
        checkbutton.connect("toggled", self.on_check_button_toggled)
        self.add(checkbutton)

    def on_check_button_toggled(self, checkbutton):
        if checkbutton.get_active():
            print("CheckButton toggled on!")
        else:
            print("CheckButton toggled off!")

window = CheckButton()
window.show_all()

Gtk.main()
```

Download: `CheckButton`

CHAPTER

FOURTEEN

RADIOBUTTON

RadioButton widgets are similar to CheckButton widgets, however they allow only one of a group of RadioButton widgets to be selected at any one time. When another RadioButton in the group is selected, the active state of the button is switched to that and removed from the previous one which was chosen.

A RadioButton is based on the *ToggleButton* widget, and inherits many of the same methods, properties and signals.

14.1 Constructor

The RadioButton can be constructed using the following:

```
radiobutton = Gtk.RadioButton(label, group, use_underline)
```

The *label* value should be set to describe the function of the RadioButton. The *group* parameter is set to the name of the first RadioButton to be included in the group. For the first RadioButton, this should be set to `None`. Subsequent RadioButton widgets should then have their group parameter set to the name of the first RadioButton. When *use_underline* is set to `True`, the character preceded by an underscore will be marked as the accelerator character.

14.2 Methods

To join a RadioButton to a group after construction use:

```
radiobutton.join_group(group)
```

The label associated with the RadioButton can be set after constructing with:

```
radiobutton.set_label(label)
```

It is good practice to use a mnemonic in the label. This requires an underscore inserted into the label (e.g. "_Cancel"). GTK+ parses the underscore and converts it into an underline beneath the following character, which the user can then access as a shortcut to the function.

    radiobutton.set_use_underline(use_underline)

To set the RadioButton as active call the method:

```
radiobutton.set_active(active)
```

When *active* is set to `True`, the RadioButton indicator will feature a dot.

The active state of the RadioButton can also be retrieved using:

```
radiobutton.get_active()
```

In some cases, the RadioButton may be set to an inconsistent state, which is used to indicate the status of other RadioButton widgets. For example, three RadioButton's may be a mix of set and unset, which leaves the fourth set as inconsistent. This can be set programatically with:

```
radiobutton.set_inconsistent(inconsistent)
```

To retrieve whether a RadioButton is set as inconsistent use:

```
radiobutton.get_inconsistent()
```

If the RadioButton is in an inconsistent state, `True` will be returned.

14.3 Signals

The commonly used signals of a RadioButton are:

```
"toggled" (radiobutton)
"group-changed" (radiobutton)
```

A `"toggled"` signal emits from the RadioButton when the mode is changed to active or inactive. When using this signal, you will need to check which RadioButton is receiving the active or inactive state. This is down to the `"toggled"` signal being emitted twice; once for the RadioButton becoming active and again for the RadioButton becoming inactive. The `"group-changed"` signal emits whenever a RadioButton changes which group it belongs to.

14.4 Example

Below is an example of a RadioButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class RadioButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        box = Gtk.Box()
        box.set_orientation(Gtk.Orientation.VERTICAL)
        box.set_spacing(5)
        self.add(box)

        radiobutton1 = Gtk.RadioButton(label="RadioButton 1")
        radiobutton1.connect("toggled", self.on_radio_button_toggled)
        box.pack_start(radiobutton1, True, True, 0)
        radiobutton2 = Gtk.RadioButton(label="RadioButton 2", group=radiobutton1)
        radiobutton2.connect("toggled", self.on_radio_button_toggled)
        box.pack_start(radiobutton2, True, True, 0)
        radiobutton3 = Gtk.RadioButton(label="RadioButton 3", group=radiobutton1)
```

(continues on next page)

(continued from previous page)

```
        radiobutton3.connect("toggled", self.on_radio_button_toggled)
        box.pack_start(radiobutton3, True, True, 0)

    def on_radio_button_toggled(self, radiobutton):
        if radiobutton.get_active():
            print("%s is active" % (radiobutton.get_label()))

window = RadioButton()
window.show_all()

Gtk.main()
```

Download: `RadioButton`

CHAPTER

FIFTEEN

LINKBUTTON

The LinkButton is similar to a Button widget, however its only function is to display website links and when clicked, open the system default browser to display the website.

15.1 Constructor

The LinkButton can be constructed using the following:

```
linkbutton = Gtk.LinkButton(uri, label)
```

The *uri* parameter should be set to the web address of the website which the LinkButton links to. The *label* parameter is optional, however is much neater when set as it displays the website name rather than link.

15.2 Methods

To set the *uri* of the LinkButton after construction use:

```
linkbutton.set_uri(uri)
```

Alternatively, the label can be set with:

```
linkbutton.set_label(label)
```

The method to retrive the *uri* from the LinkButton is:

```
linkbutton.get_uri()
```

The following method exists to check whether the link has been visited:

```
linkbutton.get_visited()
```

15.3 Signals

The commonly used signals of an LinkButton are:

```
"activate-link" (linkbutton)
```

An "activate-link" signal is emitted when the user clicks on the LinkButton. By default, clicking on the LinkButton opens a web browser but the signal can be used to cause another function with custom behaviour to be run.

15.4 Example

Below is an example of a LinkButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class LinkButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__()
        self.connect("destroy", Gtk.main_quit)

        linkbutton = Gtk.LinkButton(uri="https://programmica.com/",
                                    label="Programmica")
        self.add(linkbutton)

window = LinkButton()
window.show_all()

Gtk.main()
```

Download: LinkButton

CHAPTER

SIXTEEN

SPINBUTTON

A SpinButton provides an Entry with two associated Button widgets that allow numerical values to be entered via the keyboard, or selected by moving up or down.

16.1 Constructor

A SpinButton can be constructed using the following:

```
spinbutton = Gtk.SpinButton(adjustment, climb_rate, digits)
```

The *adjustment* parameter should be set to the name of an Adjustment object which determines the minimum and maximum values. Setting the *climb_rate* controls how fast the values change when the up or down buttons are held. The *digits* parameter is an integer value that when set controls how many numbers after the decimal point are shown.

16.2 Methods

The value within the SpinButton can be retrieved using one of the following:

```
value = spinbutton.get_value()
value = spinbutton.get_value_as_int()
```

When using `.get_value_as_int()`, the number retrieved from the SpinButton will be in integer format. Alternatively, use `.get_value()` if the number needs to be returned as a decimal.

Setting of the value within the SpinButton is possible with:

```
spinbutton.set_value(value)
```

The number of digits which will be displayed in the SpinButton can be set as:

```
spinbutton.set_digits(digits)
```

To set the Adjustment after construction:

```
spinbutton.set_adjustment(adjustment)
```

The update method can be configured with:

```
spinbutton.set_update_policy(policy)
```

When *policy* is set to `Gtk.UpdatePolicy.ALWAYS`, the SpinButton will display any content entered, even if it is not valid (i.e. out of range, alphabetical, etc). Alternatively, if `Gtk.UpdatePolicy.UPDATE_IF_VALID` is used, the SpinButton will revert to the last 'good' value if the content entered is invalid.

Control over whether non-numeric data can be entered is set with:

```
spinbutton.set_numeric(numeric)
```

The *numeric* parameter is a Boolean value which when set to `False`, allows alphabetical characters to be entered and accepted.

If a number out of range (higher than the maximum, lower than the minimum), the SpinButton can be configured to adjust the opposite limit value with:

```
spinbutton.set_wrap(wrap)
```

The *wrap* parameter should be set to `True` to enable the value wrapping feature. By default, this is set to `False` meaning that if a number out of range is entered, the SpinButton rounds to the nearest limit.

To force the SpinButton to force the value to the nearest step increment, use:

```
spinbutton.set_snap_to_ticks(snap)
```

The SpinButton can be span programmatically using:

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

The *direction* value indicates which way the SpinButton will move. This should be a constant from:

- `Gtk.Spin.STEP_FORWARD`
- `Gtk.Spin.STEP_BACKWARD`
- `Gtk.Spin.PAGE_FORWARD`
- `Gtk.Spin.PAGE_BACKWARD`
- `Gtk.Spin.HOME`
- `Gtk.Spin.END`
- `Gtk.Spin.USER_DEFINED`

The *increment* parameter is the value by which the SpinButton will change.

To manually force an update to the SpinButton:

```
spinbutton.update()
```

16.3 Signals

The commonly used signals of a SpinButton are:

```
"value-changed" (spinbutton)
```

The `"value-changed"` emits from the widget when the value contained within the SpinButton is adjusted, either via text entry or the adjusting buttons.

16.4 Example

Below is an example of a SpinButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class SpinButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        adjustment = Gtk.Adjustment(value=0,
                                    lower=-10,
                                    upper=25,
                                    step_increment=1,
                                    page_increment=5,
                                    page_size=0)
        spinbutton = Gtk.SpinButton(adjustment=adjustment)
        spinbutton.connect("value-changed", self.on_spinbutton_changed)
        self.add(spinbutton)

    def on_spinbutton_changed(self, spinbutton):
        print("SpinButton value: %i" % (spinbutton.get_value_as_int()))

window = SpinButton()
window.show_all()

Gtk.main()
```

Download: `SpinButton`

CHAPTER

SEVENTEEN

SCALEBUTTON

A ScaleButton provides a *Scale* which can be accessed via clicking a button which provides a popup menu showing the scale.

17.1 Constructor

The ScaleButton can be constructed using:

```
scalebutton = Gtk.ScaleButton(size, min_value, max_value, step, icons)
```

The *size* parameter indicates the size of the icons within the ScaleButton. The constant should be set to one of the following; `Gtk.IconSize.INVALID`, `Gtk.IconSize.MENU`, `Gtk.IconSize.SMALL_TOOLBAR`, `Gtk.IconSize.LARGE_TOOLBAR`, `Gtk.IconSize.BUTTON`, `Gtk.IconSize.DND`, or `Gtk.IconSize.DIALOG`. The *min_value*, *max_value* and, *step* values should be set to integers which indicate the minimum and maximum values on the scale, along with the increase and decrease in the value when the + or - buttons are pressed. Finally, the *icons* parameter should be a tuple, which indicates the icons shown when the scale changes.

17.2 Methods

To retrieve the value from the ScaleButton run:

```
value = scalebutton.get_value()
```

Alternatively, to set the value use:

```
scalebutton.set_value(value)
```

If an *Adjustment* is required to be used with the ScaleButton, this can be specified with:

```
scalebutton.set_adjustment(adjustment)
```

The apply icons to the ScaleButton which are adjusted when the scale changes, these can be specified using:

```
scalebutton.set_icons(icons)
```

The *icons* value is a tuple, which lists the icons that should be shown on the ScaleButton.

17.3 Example

Below is an example of a ScaleButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ScaleButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

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

        scalebutton = Gtk.ScaleButton()
        scalebutton.set_icons(("gtk-go-down", "gtk-go-up"))
        scalebutton.connect("value-changed", self.on_scale_button_changed)
        grid.attach(scalebutton, 0, 0, 1, 1)

    def on_scale_button_changed(self, scalebutton, value):
        print("ScaleButton value: %0.2f" % (value))

window = ScaleButton()
window.show_all()

Gtk.main()
```

Download: `ScaleButton`

CHAPTER

EIGHTEEN

VOLUMEBUTTON

A VolumeButton is used to control the volume within an application. It is very similar to a *ScaleButton*.

18.1 Constructor

The VolumeButton can be constructed using:

```
volumebutton = Gtk.VolumeButton()
```

18.2 Methods

To retrieve the current value from the VolumeButton use:

```
value = volumebutton.get_value()
```

Alternatively, to set a particular value run:

```
volumebutton.set_value(value)
```

The *value* argument must be an integer value between `0` and `100`.

18.3 Example

Below is an example of a VolumeButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class VolumeButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

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

(continues on next page)

(continued from previous page)

```
        volumebutton = Gtk.VolumeButton()
        volumebutton.connect("value-changed", self.on_volume_button_changed)
        grid.attach(volumebutton, 0, 0, 1, 1)

    def on_volume_button_changed(self, volumebutton, value):
        print("VolumeButton value: %0.2f" % (value))

window = VolumeButton()
window.show_all()

Gtk.main()
```

Download: `VolumeButton`

CHAPTER

NINETEEN

LOCKBUTTON

A LockButton provides a way for a user to make changes to system settings that require elevated privileges. The widget is packed with a label and icon identifying the unlock and lock actions.

19.1 Constructor

The LockButton can be constructed using the following:

```
lockbutton = Gtk.LockButton()
```

19.2 Methods

To set the permission of the LockButton use the method:

```
lockbutton.set_permission(permission)
```

The permission value can also be retrived from the LockButton via:

```
permission = lockbutton.get_permission()
```

19.3 Example

Below is an example of a LockButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk, GObject, Polkit, Gio
import os

def check_authorization(lockbutton):
    authority.check_authorization(subject, action_id, None, Polkit.
↪CheckAuthorizationFlags.ALLOW_USER_INTERACTION, cancellable, check_authorization_cb,
↪ mainloop)

def check_authorization_cb(authority, res, loop):
    try:
        result = authority.check_authorization_finish(res)
```

(continues on next page)

(continued from previous page)

```
        if result.get_is_authorized():
            print("Authorized")
        elif result.get_is_challenge():
            print("Challenge")
        else:
            print("Not authorized")
    except GObject.GError as error:
         print("Error checking authorization: %s" % error.message)

if __name__ == "__main__":
    action_id = "org.freedesktop.policykit.exec"

    mainloop = GObject.MainLoop()
    authority = Polkit.Authority.get()
    cancellable = Gio.Cancellable()
    subject = Polkit.UnixProcess.new(os.getppid())

    window = Gtk.Window()
    window.connect("destroy", Gtk.main_quit)

    lockbutton = Gtk.LockButton()
    lockbutton.connect("clicked", check_authorization)
    window.add(lockbutton)

    window.show_all()

    mainloop.run()
```

Download: `LockButton`

CHAPTER

TWENTY

SWITCH

A Switch widget provides a toggle which enables or disables depending on the position of the widget. It is commonly used to indicate the status of a hardware device.

20.1 Constructor

The Switch can be constructed using:

```
switch = Gtk.Switch()
```

20.2 Methods

To retrieve the state of the Switch as a True or False value:

```
active = switch.get_active()
```

Alternatively, to set a state on the Switch programmatically:

```
switch.set_active(active)
```

If *active* is set to True, the Switch will be in the On position.

20.3 Properties

The height of the slider can be configured with the `"slider-height"` property. The default value is 22, with allowed values greater than 22:

```
switch.set_property("slider-height", height)
```

The width can also be defined via `"slider-width"` with allowed values greater than the default of 36:

```
switch.set_property("slider-width", width)
```

20.4 Signals

The commonly use signals of a Switch are:

```
"notify::active" (switch, state)
```

The "notify::active" signal emits when the Switch is toggled to either the on or off states.

20.5 Example

Below is an example of a Switch:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Switch(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        switch = Gtk.Switch()
        switch.connect("notify::active", self.on_switch_toggled)
        self.add(switch)

    def on_switch_toggled(self, switch, state):
        if switch.get_active():
            print("Switch toggled to on")
        else:
            print("Switch toggled to off")

window = Switch()
window.show_all()

Gtk.main()
```

Download: Switch

CHAPTER

TWENTYONE

ENTRY

Entry widgets provide a way for the user to enter text. Usually they are tailored for small amounts of text such as a username, street name, or file name.

21.1 Constructor

The Entry can be constructed using:

```
entry = Gtk.Entry(entrybuffer)
```

The *entrybuffer* parameter should be set to an *EntryBuffer* if required. In most cases, it won't be needed and can be omitted.

21.2 Methods

Text can be inserted into the Entry using:

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

Using the `.set_text()` method will overwrite any existing contents in the Entry.

Alternatively, it can be inserted with:

```
entry.insert_text(text, length, position)
```

The *text* value is the string of text to be inserted. The *length* parameter is the length of the new text being inserted, however in most cases using `-1` is sufficient; as GTK+ will automatically calculate the length. Finally, the *position* parameter will specify the location in number of characters where the text will be placed.

Text can be removed from the Entry via:

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

The *start* and the *end* values indicate the range of characters to be removed.

The Entry supports returning a string of characters from a requested range:

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

Limiting the number of characters which can be typed into the Entry is set with:

```
entry.set_max_length(length)
```

To prevent editing of the Entry use:

```
entry.set_editable(editable)
```

When set to `False`, the Entry will not accept any text input.

Entry widgets are useful for receiving password input. However, it is good practice to mask the input as it is being typed to improve security:

```
entry.set_visibility(visibility)
```

When *visibility* is set to `False`, eah character will be masked with a `*`.

The mask character can be changed if required via:

```
entry.set_invisible_char(character)
```

By default, when an Entry is focused, the text within the Entry is selected. In cases where the user is not likely to want to replace all the text, an alternative function can be used to provide focus but not select the text:

```
entry.grab_focus_without_selecting()
```

In some cases, it is preferable to have the Entry perform an action when the user presses `Enter`. Commonly this would be a continue function in a dialog:

```
entry.set_activates_default(activates)
```

In some cases, it may be useful to include some placeholder text in the Entry, which indicates the purpose of the widget:

```
entry.set_placeholder_text(text)
```

A useful function for web browsers or other widgets which load content is to display a progress bar within the Entry:

```
entry.set_progress_fraction(fraction)
```

The *fraction* value is a number between `0.0` and `1.0` indicating 0% and 100% respectively.

The width of Entry in characters can be specified using the method:

```
entry.set_width_chars(width)
```

If required, icons can be placed in the Entry. There are two types; primary and secondary. The primary icon is placed on the left side of the Entry, preceding the text. Secondary icons are placed at the right-hand side of the Entry.

entry.set_icon_from_pixbuf(position, pixbuf)

The *position* value should be set to either `Gtk.EntryIconPosition.PRIMARY` or `Gtk.EntryIconPosition.SECONDARY`. The *pixbuf* value is the image to be inserted.

Icons can also be made insensitive to prevent an action:

```
entry.set_icon_sensitive(position, sensitive)
```

When *sensitive* is set to `False`, the icon specified will appear greyed-out.

If an icon has been specified, tooltip text can also be set describing the function of the icon.

entry.set_icon_tooltip_text(position, tooltip) entry.set_icon_tooltip_markup(position, tooltip)

The `.set_icon_tooltip_text()` takes plain text only. Alternatively, styled text can be specified using `.set_icon_tooltip_markup()`.

Entry widgets also support input types. This describes the function of the widget, and is useful as an accessibility function.

entry.set_input_purpose(purpose)

The *purpose* should be set to one of the following values:

- `Gtk.InputPurpose.FREE_FORM`
- `Gtk.InputPurpose.ALPHA`
- `Gtk.InputPurpose.DIGITS`
- `Gtk.InputPurpose.NUMBER`
- `Gtk.InputPurpose.PHONE`
- `Gtk.InputPurpose.URL`
- `Gtk.InputPurpose.EMAIL`
- `Gtk.InputPurpose.NAME`
- `Gtk.InputPurpose.PASSWORD`
- `Gtk.InputPurpose.PIN`

Hints are also available which allow the input to be tailored as required. This is done via:

```
entry.set_input_hints(hints)
```

The *hints* value can be set to one of:

- `Gtk.InputHint.NONE` - no special behaviour.
- `Gtk.InputHint.SPELLCHECK` - suggest spell checking for errors.
- `Gtk.InputHint.NO_SPELLCHECK` - suggest no spell checking takes place.
- `Gtk.InputHint.WORD_COMPLETION` - suggestion word completion should be used.
- `Gtk.InputHint.LOWERCASE` - suggest to lowercase all text.
- `Gtk.InputHint.UPPERCASE_CHARS` - suggest to capitalise all text.
- `Gtk.InputHint.UPPERCASE_WORDS` - suggest to capitalise first letter in all words.
- `Gtk.InputHint.UPPERCASE_SENTENCES` - suggest to capitalise first word in each sentence.
- `Gtk.InputHint.INHIBIT_OSK` - suggest onscreen keyboard not be shown.
- `Gtk.InputHint.VERTICAL_WRITING` - text is vertical.

21.3 Signals

The commonly used signals of an Entry are:

```
"changed" (entry)
"activate" (entry)
"icon-press" (entry, icon_pos, event)
"icon-release" (entry, icon_pos, event)
"backspace" (entry)
"populate-popup" (entry, menu)
```

The "changed" signal is emitted whenever there is a change to the Entry. The "activate" signal is emitted whenever the Enter button is pressed when the Entry has focus. An "icon-press" signal is emitted when any mouse button is pressed down on either the primary or secondary icons. An "icon-release" is much the same as "icon-press" except emits when the mouse button is released. The "backspace" signal emits whenever the user presses the Backspace key. Using "populate-popup" allows items to be dynamically places in the context menu of the Entry prior to it being displayed.

21.4 Examples

Below is an example of a Entry:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Entry(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        entry = Gtk.Entry()
        entry.set_placeholder_text("Entry text here...")
        entry.connect("activate", self.on_entry_activated)
        self.add(entry)

    def on_entry_activated(self, entry):
        print("Entry text: %s" % (entry.get_text()))

window = Entry()
window.show_all()

Gtk.main()
```

Download: Entry

CHAPTER

TWENTYTWO

ENTRYBUFFER

An EntryBuffer contains text which can be displayed inside an *Entry*. Its purpose is to allow that text to be shared between multiple Entry widgets.

Note: To use an Entry widget, an EntryBuffer object is not required and in most cases will not be needed.

22.1 Constructor

The EntryBuffer can be constructed using the following:

```
entrybuffer = Gtk.EntryBuffer(text)
```

The *text* string may be omitted, however any text entered will be displayed in Entry widgets connected to the EntryBuffer.

22.2 Methods

To retrieve text from the EntryBuffer, use the method:

```
entrybuffer.get_text()
```

To set text on the EntryBuffer after construction:

```
entrybuffer.set_text(text, n_chars)
```

The *text* parameter should be set to the text which is to be displayed in the EntryBuffer. The *n_chars* parameter is an integer value limiting the number of characters which can be entered. If there is to be no limit, use `-1`.

To insert text into the EntryBuffer at a specific position, the following can be used:

```
entrybuffer.insert_text(position, text)
```

The *position* value is the number of characters along the text should be inserted at with `0` designated as the beginning of the EntryBuffer. The *text* parameter should be the string of text to be inserted.

Deletion of text from the EntryBuffer can also be done with:

```
entrybuffer.delete_text(position, n_chars)
```

The *position* parameter should be the starting position of the cursor. The *n_chars* parameter indicates the number of characters from the position to be deleted. When set to a negative number, all characters up to the cursor are deleted.

Retrieval of the number of characters currently stored in the EntryBuffer is achieved by using:

```
entrybuffer.get_length()
```

Setting the maximum number of characters that the EntryBuffer can accept construction can be completed with:

```
entrybuffer.set_max_length(length)
```

To check the maximum length the EntryBuffer can accept:

```
entrybuffer.get_max_length()
```

22.3 Signals

The commonly used signals of an EntryBuffer are:

```
"inserted-text" (buffer, position, chars, n_chars)
"deleted-text" (buffer, position, n_chars)
```

The `"inserted-text"` and `"deleted-text"` signals are activated when textual values are added to or deleted from the EntryBuffer. Both pass the EntryBuffer, the position at which the change occurred, and the number of characters that were added and removed. The `"inserted-text"` signal however has an extra value which passes the string of chracters which were inserted.

22.4 Example

Below is an example of a EntryBuffer:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class EntryBuffer(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        box = Gtk.Box()
        box.set_orientation(Gtk.Orientation.VERTICAL)
        self.add(box)

        entrybuffer = Gtk.EntryBuffer()
        entrybuffer.set_text("Text in EntryBuffer", -1)

        entry = Gtk.Entry()
        entry.set_buffer(entrybuffer)
        box.pack_start(entry, True, True, 0)
        entry = Gtk.Entry()
        entry.set_buffer(entrybuffer)
        box.pack_start(entry, True, True, 0)
```

(continues on next page)

(continued from previous page)

```
        entry = Gtk.Entry()
        entry.set_buffer(entrybuffer)
        box.pack_start(entry, True, True, 0)

window = EntryBuffer()
window.show_all()

Gtk.main()
```

Download: `EntryBuffer`

CHAPTER

TWENTYTHREE

ENTRYCOMPLETION

An EntryCompletion can be attached to an Entry widget to provide suggestions which match the characters entered.

23.1 Constructor

The EntryCompletion can be constructed using the following:

```
entrycompletion = Gtk.EntryCompletion()
```

23.2 Methods

Once the EntryCompletion has been constructed, a model needs to be attached. This is used to store the potential values which can be matched:

```
entrycompletion.set_model(model)
```

The *model* should be the name of a ListStore object.

The text column must also be set which refers to which column within the ListStore the potentially matches should be listed:

```
entrycompletion.set_text_column(column)
```

To prevent completion lookups occuring before a certain number have characters have been entered, the following method can be used:

```
entrycompletion.set_minimum_key_length(length)
```

The *length* parameter should be set to an integer value.

To cycle through the position completions within the Entry:

```
entrycompletion.set_inline_completion(inline)
```

When *inline* is set to `True`, the user can cycle through the matching completions by tabbing.

Alternatively, there is an option to provide a list of completions popped up in a popup window with:

```
entrycompletion.set_popup_completion(popup)
```

23.3 Example

Below is an example of a EntryCompletion:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class EntryCompletion(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

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

        liststore = Gtk.ListStore(str)

        for item in ["Andrew", "Natalie", "Mark", "David", "Daniel", "Anita", "Matthew
↪"]:
            liststore.append([item])

        self.entrycompletion = Gtk.EntryCompletion()
        self.entrycompletion.set_model(liststore)
        self.entrycompletion.set_text_column(0)

        entry = Gtk.Entry()
        entry.set_completion(self.entrycompletion)
        grid.attach(entry, 0, 0, 1, 1)

        radiobuttonPopup = Gtk.RadioButton("Popup Completion")
        radiobuttonPopup.mode = 0
        radiobuttonPopup.connect("toggled", self.on_radiobutton_toggled)
        grid.attach(radiobuttonPopup, 0, 1, 1, 1)
        radiobuttonInline = Gtk.RadioButton("Inline Completion")
        radiobuttonInline.mode = 1
        radiobuttonInline.join_group(radiobuttonPopup)
        radiobuttonInline.connect("toggled", self.on_radiobutton_toggled)
        grid.attach(radiobuttonInline, 0, 2, 1, 1)

    def on_radiobutton_toggled(self, radiobutton):
        if radiobutton.get_active():
            if radiobutton.mode == 0:
                self.entrycompletion.set_popup_completion(True)
                self.entrycompletion.set_inline_completion(False)
            elif radiobutton.mode == 1:
                self.entrycompletion.set_popup_completion(False)
                self.entrycompletion.set_inline_completion(True)


window = EntryCompletion()
window.show_all()

Gtk.main()
```

Download: `EntryCompletion`

CHAPTER

TWENTYFOUR

SEARCHENTRY

The SearchEntry widget provides a tailored interface for searching. Essentially, it is a stardard *Entry* with a find icon when the search field is empty, that then changes to a clear icon when text has been entered. Additional signals are also available which relate to the search functionality.

24.1 Constructor

The SearchEntry can be constructed using the following:

```
searchentry = Gtk.SearchEntry()
```

24.2 Methods

The text can be retrieved from the SearchEntry:

```
searchentry.get_text()
```

Text can also be added to the SearchEntry with the method:

```
searchentry.set_text()
```

Placeholder text can be added to the SearchEntry to describe the function of the widget:

```
searchentry.set_placeholder_text(text)
```

To initiate a default action when the user presses `Enter` or `Return`, the method used is:

```
searchentry.set_activates_default(activates)
```

Typically, the SearchEntry would only activate a default function when placed in a *Dialog*.

24.3 Signals

The commonly used signals of a SearchEntry are:

```
"search-changed" (searchentry)
"next-match" (searchentry)
"previous-match" (searchentry)
"stop-search" (searchentry)
```

The "search-changed" signal emits each time the user enters a character into the search field after a 150 millisecond delay. The "next-match" and "previous-match" signals emit when the user moves between next and previous matches for the search string. Finally, the "stop-search" is emitted when the user stops a search via keyboard input.

24.4 Example

Below is an example of a SearchEntry:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class SearchEntry(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title("SearchEntry")
        self.connect("destroy", Gtk.main_quit)

        searchentry = Gtk.SearchEntry()
        searchentry.connect("activate", self.on_search_activated)
        self.add(searchentry)

    def on_search_activated(self, searchentry):
        print("SearchEntry text: %s" % (searchentry.get_text()))

window = SearchEntry()
window.show_all()

Gtk.main()
```

Download: SearchEntry

CHAPTER

TWENTYFIVE

SEARCHBAR

The SearchBar widget provides an animated container, which provides an *Entry* or *SearchEntry* for the user.

25.1 Constructor

The SearchBar is constructed using the following:

```
searchbar = Gtk.SearchBar()
```

25.2 Methods

Items can be added to the SearchBar using:

```
searchbar.add(child)
```

Most commonly the child should be a SearchEntry, or a standard Entry widget.

An Entry or SearchEntry needs to be connected to the SearchBar with:

```
searchbar.connect_entry(entry)
```

To display whether the SearchBar is visible to the user, call:

```
searchbar.set_search_mode(search_mode)
```

If the *search_mode* parameter is `True`, the SearchBar will show. When set to `False`, it will be hidden again.

To configure whether a close button is shown on the Search call:

```
searchbar.set_show_close_button(show_button)
```

If *show_button* is `True`, the SearchBar will have a close button. The default however is `False`, and the close button is not shown.

25.3 Example

Below is an example of a SearchBar:

```
#!/usr/bin/env python3

from gi.repository import Gtk, Gdk

class SearchBar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(250, -1)
        self.set_title("SearchBar")
        self.connect("key-press-event", self.on_key_event)
        self.connect("destroy", Gtk.main_quit)

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

        label = Gtk.Label("Press Control+F to initiate find")
        grid.attach(label, 0, 0, 1, 1)

        self.searchbar = Gtk.SearchBar()
        grid.attach(self.searchbar, 0, 1, 1, 1)

        searchentry = Gtk.SearchEntry()
        self.searchbar.connect_entry(searchentry)
        self.searchbar.add(searchentry)

    def on_key_event(self, widget, event):
        shortcut = Gtk.accelerator_get_label(event.keyval, event.state)

        if shortcut in ("Ctrl+F", "Ctrl+Mod2+F"):
            if self.searchbar.get_search_mode():
                self.searchbar.set_search_mode(False)
            else:
                self.searchbar.set_search_mode(True)

window = SearchBar()
window.show_all()

Gtk.main()
```

Download: `SearchBar`

CHAPTER

TWENTYSIX

IMAGE

The Image widget is able to display a variety of Image types within an application, and supports a number of formats including PNG, JPG, BMP, SVG, GIF, XPM.

26.1 Constructor

The Image can be constructed using the following:

```
image = Gtk.Image()
```

26.2 Methods

After construction, there are a number of methods which can be used to display different image types. The common ones are:

```
image.set_from_file(file_path)
image.set_from_pixbuf(pixbuf)
image.set_from_resource(resource_path)
```

The pixel size of the Image object can be set and retrieved with the methods:

```
image.set_pixel_size(size)
size = image.get_pixel_size()
```

To clear the graphic from the Image:

```
image.clear()
```

Retrieval of the type of image currently held in the Image object can be done with:

```
storage_type = image.get_storage_type()
```

26.3 Example

Below is an example of a Image:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Image(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        image = Gtk.Image()
        image.set_from_file("../_resources/gtk.png")
        self.add(image)

window = Image()
window.show_all()

Gtk.main()
```

Download: `Image`

CHAPTER

TWENTYSEVEN

SPINNER

The Spinner widget is used to show activity in progress. Usually, the Spinner is used when it is not known how long the action may take. A common example of its use is to indicate the loading of a web page.

27.1 Constructor

The Spinner can be constructed using:

```
spinner = Gtk.Spinner()
```

27.2 Methods

The two methods which allow starting and stopping of the Spinner are:

```
spinner.start()
spinner.stop()
```

27.3 Properties

To check whether the Spinner is active, the property call is:

```
active = spinner.get_property("active")
```

If the Spinner is running, `True` is returned.

Whether the Spinner is running or not is able to be set using:

```
spinner.set_property("active", active)
```

When *active* is set to `True`, the Spinner animation begins.

27.4 Examples

Below is an example of a Spinner:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Spinner(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        grid = Gtk.Grid()
        grid.set_row_spacing(5)
        grid.set_column_spacing(5)
        self.add(grid)

        self.spinner = Gtk.Spinner()
        self.spinner.set_vexpand(True)
        self.spinner.set_hexpand(True)
        grid.attach(self.spinner, 0, 0, 2, 1)

        buttonStart = Gtk.Button("Start")
        buttonStart.connect("clicked", self.on_start_clicked)
        grid.attach(buttonStart, 0, 1, 1, 1)

        buttonStop = Gtk.Button("Stop")
        buttonStop.connect("clicked", self.on_stop_clicked)
        grid.attach(buttonStop, 1, 1, 1, 1)

    def on_start_clicked(self, button):
        self.spinner.start()

    def on_stop_clicked(self, button):
        self.spinner.stop()

window = Spinner()
window.show_all()

Gtk.main()
```

Download: `Spinner`

CHAPTER

TWENTYEIGHT

INFOBAR

An InfoBar provides an in-window method of displaying messages to the user. These can be warnings, errors, questions, information or other custom-made messages.

28.1 Constructor

The InfoBar can be constructed using the following:

```
infobar = Gtk.InfoBar()
```

28.2 Methods

After construction, configuration of the message type may be needed. The InfoBar defaults to `Gtk.MessageType.INFO`, however it is also possible to display `Gtk.MessageType.WARNING`, `Gtk.MessageType.ERROR`, `Gtk.MessageType.QUESTION` or `Gtk.MessageType.OTHER` with:

```
infobar.set_message_type(message_type)
```

Buttons can be added to the InfoBar to allow different responses to the message. These can be added with:

```
infobar.add_button(text, response_id)
```

The *text* value in the first method is a string of text which should be displayed on the button. The *response_id* parameter in both methods can be set to one of the following:

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

- `Gtk.ResponseType.HELP`

Alternatively, multiple buttons can be added to the InfoBar via:

```
infobar.add_buttons(text, response_id, ...)
```

For every button text added to the list, it must also have an appropriate *response_id*.

When an InfoBar is no longer required, it is recommended to hide it from view with:

```
infobar.hide()
```

If multiple buttons are in use, it may be useful to set one as the default response. This allows the user to press Enter without any other action to run the default action:

```
infobar.set_default_response(response_id)
```

The *response_id* specified here should match one of those specified when using `.add_button()`.

Another useful feature is to be able to mark a button within the InfoBar as not sensitive. This can be achieved with:

```
infobar.set_response_sensitive(response_id, setting)
```

The *response_id* should be set to one of those specified when using `.add_button()`. The setting will be a `True` or `False` value, with `False` setting the button as insensitive (or greyed-out).

To configure whether a close button is displayed on the InfoBar, call:

```
infobar.set_show_close_button(show_button)
```

When *show_button* is set to `True`, a button is shown which allows the InfoBar to be closed. When the button is pressed, the response `Gtk.ResponseType.CLOSE` is emitted.

The MessageDialog is constructed using several predefined *Box* widgets which give the shape of the dialog. The *content_area* is the place where you place widgets. The *action_area* is the place where buttons and other actionable widgets are place. Both can be retrieved with the methods:

```
infobar.get_content_area()
infobar.get_action_area()
```

28.3 Signals

The commonly used signals of a InfoBar are:

```
"close" (infobar)
"response" (infobar, response_id)
```

The `"close"` signal emits from the InfoBar when the `Escape` key is pressed. A `"response"` signal is emitted whenever a button within the InfoBar is clicked. The *response_id* varies depending on which action the user takes.

28.4 Example

Below is an example of an InfoBar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class InfoBar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        grid = Gtk.Grid()
        grid.set_row_spacing(5)
        self.add(grid)

        self.infobar = Gtk.InfoBar()
        self.infobar.set_show_close_button(True)
        self.infobar.connect("response", self.on_infobar_response)
        grid.attach(self.infobar, 0, 0, 1, 1)

        label = Gtk.Label("InfoBar content string.")
        content = self.infobar.get_content_area()
        content.add(label)

        buttonbox = Gtk.ButtonBox()
        grid.attach(buttonbox, 0, 1, 1, 1)

        buttonInformation = Gtk.Button(label="Information")
        buttonInformation.message_type = Gtk.MessageType.INFO
        buttonInformation.connect("clicked", self.on_button_clicked)
        buttonbox.add(buttonInformation)
        buttonQuestion = Gtk.Button(label="Question")
        buttonQuestion.message_type = Gtk.MessageType.QUESTION
        buttonQuestion.connect("clicked", self.on_button_clicked)
        buttonbox.add(buttonQuestion)
        buttonWarning = Gtk.Button(label="Warning")
        buttonWarning.message_type = Gtk.MessageType.WARNING
        buttonWarning.connect("clicked", self.on_button_clicked)
        buttonbox.add(buttonWarning)
        buttonError = Gtk.Button(label="Error")
        buttonError.message_type = Gtk.MessageType.ERROR
        buttonError.connect("clicked", self.on_button_clicked)
        buttonbox.add(buttonError)
        buttonOther = Gtk.Button(label="Other")
        buttonOther.message_type = Gtk.MessageType.OTHER
        buttonOther.connect("clicked", self.on_button_clicked)
        buttonbox.add(buttonOther)

    def on_button_clicked(self, button):
        self.infobar.set_message_type(button.message_type)
        self.infobar.show()

    def on_infobar_response(self, infobar, respose_id):
        self.infobar.hide()

window = InfoBar()
window.show_all()

Gtk.main()
```

Download: `InfoBar`

CHAPTER

TWENTYNINE

ACTIONBAR

The ActionBar widget provides a full width bar for widgets such as *Button* and *Label* widgets. It is usually placed below the application content and is commonly used in place of a *Statusbar*.

29.1 Constructor

The ActionBar is constructed using:

```
actionbar = Gtk.ActionBar()
```

29.2 Methods

Items can be packed into the container at the start (left) or end (right):

```
actionbar.pack_start(child)
actionbar.pack_end(child)
```

If required, items can also be placed in the center:

```
actionbar.set_center_widget(child)
```

The center widget can also be retrieved via:

```
actionbar.get_center_widget()
```

29.3 Properties

The property items available for use with the ActionBar are:

- `"pack-type"` - can be set to either `Gtk.PackType.START` or `Gtk.Pack_type.END` for left or right placement of child widgets.
- `"position"` - specifies the index of the child in the ActionBar.

29.4 Example

Below is an example of an ActionBar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ActionBar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

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

        label = Gtk.Label()
        label.set_vexpand(True)
        grid.attach(label, 0, 0, 1, 1)

        actionbar = Gtk.ActionBar()
        actionbar.set_hexpand(True)
        grid.attach(actionbar, 0, 1, 1, 1)

        button = Gtk.Button("Cut")
        actionbar.pack_start(button)
        button = Gtk.Button("Copy")
        actionbar.pack_start(button)
        button = Gtk.Button("Paste")
        actionbar.pack_end(button)

    def run(self):
        self.show_all()

window = ActionBar()
window.run()

Gtk.main()
```

Download: `ActionBar`

CHAPTER

THIRTY

HEADERBAR

A HeaderBar allows child widgets to be inserted into it, and centered within. It also allows a title to be shown, and also centered with respect to the width of the HeaderBar.

30.1 Constructor

The HeaderBar is constructed using the call:

```
headerbar = Gtk.HeaderBar()
```

30.2 Methods

To set the title on the HeaderBar use:

```
headerbar.set_title(title)
```

The title can also be retrieved with:

```
title = headerbar.get_title()
```

HeaderBar widgets also support subtitles and would be used to provide information to the user about the current view.

headerbar.set_subtitle(subtitle)

If required, other widgets can be set as the title with the method:

```
headerbar.set_custom_title(child)
```

Items can be packed into the HeaderBar with the two methods:

```
headerbar.pack_start(child)
headerbar.pack_end(child)
```

The close button within the HeaderBar can be shown or hidden with:

```
headerbar.set_show_close_button(show_button)
```

The *show_button* value should be set to `True` or `False` depending on whether the object is shown or not.

30.3 Example

Below is an example of an HeaderBar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class HeaderBar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(-1, 200)
        self.connect("destroy", Gtk.main_quit)

        headerbar = Gtk.HeaderBar()
        headerbar.set_title("HeaderBar Example")
        headerbar.set_subtitle("HeaderBar Subtitle")
        headerbar.set_show_close_button(True)
        self.set_titlebar(headerbar)

        button = Gtk.Button("Open")
        headerbar.pack_start(button)

window = HeaderBar()
window.show_all()

Gtk.main()
```

Download: `HeaderBar`

CHAPTER

THIRTYONE

STATUSBAR

A Statusbar is positioned at the bottom of some application windows to provide status messages and information about an applications current process. For example, it can be used to indicate the line and column number within a text editor or which website a hyperlink directs to in a web browser.

Messages on a Statusbar are stored in a stack, with the first message pushed on to the Statusbar being the last message to be popped from it.

Note: Statusbar widgets should be used to display messages of low-importance. If a user must see a message, a *MessageDialog* or *InfoBar* is the recommended widget to use.

31.1 Constructor

The Statusbar can be constructed using:

```
statusbar = Gtk.Statusbar()
```

31.2 Methods

Before messages can be displayed on the Statusbar, a context identifier needs to be retrieved. This context identifier is a string which identifies particular message types, for example; errors and warnings. This can be retrived with:

```
statusbar.get_context_id(context)
```

The *context* parameter is simply a string describing the context (purpose) of the message. The method returns the context id which is used to push, pop and remove messages.

To push a message onto the Statusbar call:

```
statusbar.push(context, text)
```

When calling the `.push()` method, a message id is returned. This value is unique and identifies a particular message within the Statusbar.

Messages can be popped from the list with:

```
statusbar.pop(context)
```

Alternatively, if a message is to be completely removed from the Statusbar stack, call:

```
statusbar.remove(context, message)
```

The *context* is the one specified before calling the `.push()` method. The *message* parameter takes the message id for the message to be removed, and must be specified.

Alternatively, to remove all messages within a particular context use:

```
statusbar.remove_all(context)
```

In some cases, it can be useful to add widgets such as ComboBox widgets to a Statusbar to provide quick setting selection as well as providing information. To retrieve the container, which can then be added to use:

```
statusbar.get_message_area()
```

The `.get_message_area()` method returns a *Box* with horizontal orientation.

31.3 Signals

The commonly used signals of an Statusbar are:

```
"text-pushed" (context, text)
"text-popped" (context, text)
```

The *text-pushed* and *text-popped* signals emit when a message is pushed to or popped from the Statusbar. Both signals return the context id of the message and the textual content.

31.4 Example

Below is an example of a Statusbar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Statusbar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        grid = Gtk.Grid()
        grid.set_column_spacing(5)
        self.add(grid)

        buttonPush = Gtk.Button("Push")
        buttonPush.connect("clicked", self.on_push_clicked)
        grid.attach(buttonPush, 0, 0, 1, 1)

        buttonPop = Gtk.Button("Pop")
        buttonPop.connect("clicked", self.on_pop_clicked)
        grid.attach(buttonPop, 1, 0, 1, 1)

        buttonRemove = Gtk.Button("Remove All")
        buttonRemove.connect("clicked", self.on_remove_all_clicked)
```

(continues on next page)

(continued from previous page)

```
        grid.attach(buttonRemove, 2, 0, 1, 1)

        self.statusbar = Gtk.Statusbar()
        self.context = self.statusbar.get_context_id("example")
        grid.attach(self.statusbar, 0, 1, 3, 1)

        self.count = 0

    def on_push_clicked(self, button):
        self.count += 1

        message = "Message number %i" % (self.count)
        self.statusbar.push(self.context, message)

    def on_pop_clicked(self, button):
        self.statusbar.pop(self.context)

    def on_remove_all_clicked(self, button):
        self.statusbar.remove_all(self.context)

window = Statusbar()
window.show_all()

Gtk.main()
```

Download: `Statusbar`

CHAPTER

THIRTYTWO

SEPARATOR

A Separator widget is an ornamental widget which is used to split or group content within an interface. The Separator is a line drawn on the interface with a shadow to make it appear sunken.

32.1 Constructor

The Separator can be constructed using:

```
separator = Gtk.Separator(orientation)
```

The *orientation* parameter should be set to either `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL` depending on the requirements. If no orientation is set, the default orientation used is set vertically.

32.2 Methods

The orientation can also be set after construction with:

```
seperator.set_orientation(orientation)
```

32.3 Examples

Below is an example of a Separator:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Separator(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(400, 200)
        self.connect("destroy", Gtk.main_quit)

        box = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=5)
        box.set_homogeneous(True)
        self.add(box)

        separator = Gtk.Separator(orientation=Gtk.Orientation.VERTICAL)
```

(continues on next page)

(continued from previous page)

```
        box.pack_start(separator, True, True, 0)

        separator = Gtk.Separator(orientation=Gtk.Orientation.HORIZONTAL)
        box.pack_start(separator, True, True, 0)

window = Separator()
window.show_all()

Gtk.main()
```

Download: `Separator`

CHAPTER

THIRTYTHREE

ACCELLABEL

An AccelLabel is used to display an Accelerator which displays a keyboard combination which can be used to activate a described function.

33.1 Constructor

The AccelLabel can be constructed using the following:

```
accellabel = Gtk.AccelLabel(label)
```

A *label* should be specified for the Label which indicates the function of the accelerator.

33.2 Methods

To monitor a particular widget for an accelerator use:

```
accellabel.set_accel_widget(accel_widget)
```

The *accel_widget* value should be set to the name of a widget which is to be monitored for an accelerator. When one is applied to a child widget the AccelLabel is updated to display the accelerator combination.

The associated accelerator widget is fetchable with:

```
accellabel.get_accel_widget()
```

An accelerator can also be specified by using:

```
accellabel.set_accel(key, modifier)
```

The *key* parameter indicates the accelerator key to use, with an integer value accepted. The *modifier* sets the key such as `Control` or `Alt` and should be set to `None` if not required, or:

- `Gdk.ModifierType.CONTROL_MASK` - the Control key.
- `Gdk.ModifierType.SHIFT_MASK` - the Shift key.
- `Gdk.ModifierType.MOD1_MASK` - typically the Alt key.
- `Gdk.ModifierType.LOCK_MASK` - a lock key such as the Caps or Num lock.
- `Gdk.ModifierType.BUTTON1_MASK` - mouse button 1.
- `Gdk.ModifierType.BUTTON2_MASK` - mouse button 2.

- `Gdk.ModifierType.BUTTON3_MASK` - mouse button 3.

An accelerator can also be retrieved via:

```
accellabel.get_accel()
```

33.3 Example

Below is an example of a AccelLabel:

```
#!/usr/bin/env python3

from gi.repository import Gtk, Gdk

def button_clicked(widget):
    print("Save button clicked")

window = Gtk.Window()
window.set_default_size(200, -1)
window.connect("destroy", Gtk.main_quit)

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

accelgroup = Gtk.AccelGroup()
window.add_accel_group(accelgroup)

accellabel = Gtk.AccelLabel("Button accelerator:")
accellabel.set_hexpand(True)
grid.attach(accellabel, 0, 0, 2, 1)

button = Gtk.Button("Save")
button.add_accelerator("clicked",
                       accelgroup,
                       Gdk.keyval_from_name("s"),
                       Gdk.ModifierType.CONTROL_MASK,
                       Gtk.AccelFlags.VISIBLE)
button.connect("clicked", button_clicked)
accellabel.set_accel_widget(button)
grid.attach(button, 0, 1, 2, 1)

window.show_all()

Gtk.main()
```

Download: `AppChooserButton`

CHAPTER

THIRTYFOUR

ACCELGROUP

An AccelGroup represents a group of accelerators which provide access to functions via the keyboard.

34.1 Constructor

The AccelGroup can be constructed using:

```
accelgroup = Gtk.AccelGroup()
```

34.2 Methods

An accelerator is connected to the AccelGroup using the call:

```
accelgroup.connect(accel_key, modifiers, flags, closure)
```

A previously connected accelerator can also be disconnected by:

```
accelgroup.disconnect(closure)
```

To allow or prevent changes to the Accelerator's within the AccelGroup use:

```
accelgroup.lock()
accelgroup.unlock()
```

The AccelGroup can also be checked to see whether it is locked using:

```
accelgroup.get_is_locked()
```

If `True` is returned from the method, no accelerators may be added to the AccelGroup.

34.3 Example

For an example of an AccelGroup, see the *AccelLabel* page.

CHAPTER

THIRTYFIVE

CALENDAR

A Calendar widget allows for displaying and retrival of date information. It can be configured in many ways and is a relatively powerful widget.

35.1 Constructor

The Calendar can be constructed using the following:

```
calendar = Gtk.Calendar()
```

35.2 Methods

The date can be retrieved from the Calendar using:

```
calendar.get_date()
```

The *date* will be returned in comma-seperated format. The day value will be between 1 and 31, the month will be between 0 and 11 and the date will be in four-digit number format.

To select a specific day on the Calendar call:

```
calendar.select_day(day)
```

Alternatively, to set a month and year call:

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

To configure the view of the Calendar, the following method can be used:

```
calendar.set_display_options(flags)
```

The *flags* parameter can take the following parameters; `Gtk.CalendarDisplayOptions.SHOW_HEADING` configures whether the month and year should be displayed, `Gtk.CalendarDisplayOptions.SHOW_DAY_NAMES` specifies whether the three letter day description should be present, `Gtk.CalendarDisplayOptions.NO_MONTH_CHANGE` prevents the user from changing the month. `Gtk.CalendarDisplayOptions.SHOW_WEEK_NUMBERS` sets the calendar to display the week number down the left-side of the Calendar and `Gtk.CalendarDisplayOptions.SHOW_DETAILS` sets the calendar to show an indicator as opposed to the full details text. When multiple flags are required, they should be a seperated by a 'l' character.

The defined display options can be retrieved from the Calendar using:

```
calendar.get_display_options()
```

A day can be marked or unmarked by specifying the day integer:

```
calendar.mark_day(day)
calendar.unmark_day(day)
```

To check whether a day is marked, use:

```
calendar.get_day_is_marked(day)
```

All day marks can be cleared from the Calendar via the method:

```
calendar.clear_marks()
```

Extra information can be displayed on the Calendar, such as appointments, birthdays, holidays, etc. This is done by specifying a function which handles the detail values. This is specified with:

```
calendar.set_detail_func(function, data, destroy)
```

Detail information is only shown on the Calendar when the `Gtk.CalendarDisplayOptions.SHOW_DETAILS` parameter is set.

Sizing values can be set with the methods:

```
calendar.set_detail_width_chars(width)
calendar.set_detail_height_rows(height)
```

The *width* value defines the number of pixels permitted for displaying of detail information. The *height* integer sets the number of rows of information to show.

35.3 Signals

The commonly used signals of a Calendar are:

```
"day-selected" (calendar)
"day-selected-double-click" (calendar)
"month-changed" (calendar)
"next-month" (calendar)
"next-year" (calendar)
"prev-month" (calendar)
"prev-year" (calendar)
```

The `"day-selected"` and `"day-selected-double-click"` signals emit from the Calendar when the user clicks or double clicks on a date. A `"month-changed"` signal is emitted when the user changes months and year. The signals `"next-month"`, `"next-year"`, `"prev-month"`, and `"prev-year"` emit whenever the users moves back or forward on a month or year basis.

35.4 Example

Below is an example of a Calendar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Calendar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title("Calendar")
        self.connect("destroy", Gtk.main_quit)

        hbox = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=2)
        self.add(hbox)

        self.calendar = Gtk.Calendar()
        self.calendar.connect("day-selected-double-click", self.on_date_selected)
        hbox.pack_start(self.calendar, True, True, 0)

        vbox = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=2)
        hbox.pack_start(vbox, False, False, 0)

        checkbuttonHeading = Gtk.CheckButton(label="Show Heading")
        checkbuttonHeading.set_active(True)
        checkbuttonHeading.connect("toggled", self.on_show_heading_change)
        vbox.pack_start(checkbuttonHeading, False, False, 0)

        checkbuttonDayNames = Gtk.CheckButton(label="Show Day Names")
        checkbuttonDayNames.set_active(True)
        checkbuttonDayNames.connect("toggled", self.on_show_days_change)
        vbox.pack_start(checkbuttonDayNames, False, False, 0)

        checkbuttonPreventChange = Gtk.CheckButton(label="Prevent Month/Year Change")
        checkbuttonPreventChange.connect("toggled", self.on_prevent_month_change)
        vbox.pack_start(checkbuttonPreventChange, False, False, 0)

        checkbuttonShowWeeks = Gtk.CheckButton(label="Show Week Numbers")
        checkbuttonShowWeeks.connect("toggled", self.on_show_weeks_change)
        vbox.pack_start(checkbuttonShowWeeks, False, False, 0)

    def on_date_selected(self, calendar):
        year, month, day = self.calendar.get_date()
        month += 1

        print("Date selected: %i/%i/%i" % (year, month, day))

    def on_show_heading_change(self, checkbutton):
        self.calendar.set_property("show-heading", checkbutton.get_active())

    def on_show_days_change(self, checkbutton):
        self.calendar.set_property("show-day-names", checkbutton.get_active())

    def on_prevent_month_change(self, checkbutton):
        self.calendar.set_property("no-month-change", checkbutton.get_active())

    def on_show_weeks_change(self, checkbutton):
        self.calendar.set_property("show-week-numbers", checkbutton.get_active())

window = Calendar()
```

(continues on next page)

(continued from previous page)

```
window.show_all()

Gtk.main()
```

Download: `Calendar`

CHAPTER

THIRTYSIX

EVENTBOX

An EventBox is an invisible widget which can be used to detect events which occur within it. For example, it can be used to make a Label clickable.

36.1 Constructor

The EventBox can be constructed using the following:

```
eventbox = Gtk.EventBox()
```

36.2 Methods

A child widget should be added to the EventBox with:

```
eventbox.add(child)
```

Child widgets can also be removed by calling:

```
eventbox.remove(child)
```

In some cases, it may be necessary to configure the position of the EventBox relative to the child.

eventbox.set_above_child(above_child)

If *above_child* is `True`, the EventBox will receive all input. When set to the `False`, the EventBox will receive events after they have gone to the child widget.

36.3 Signals

The EventBox can take a large number of events with the most common being:

```
"event"
"button-press-event"
"button-release-event"
"scroll-event"
"query-tooltip"
"show"
"hide"
```

(continues on next page)

(continued from previous page)

```
"enter-notify-event"
"leave-notify-event"
```

36.4 Example

Below is an example of an EventBox:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def event(eventbox, event):
    print("Event: %s" % event)

def event_press(eventbox, event):
    print("Button Press Event: %s" % event)

def event_release(eventbox, event):
    print("Button Release Event: %s" % event)

window = Gtk.Window()
window.set_default_size(200, 200)
window.connect("destroy", Gtk.main_quit)

eventbox = Gtk.EventBox()
eventbox.connect("event", event)
eventbox.connect("button-press-event", event_press)
eventbox.connect("button-release-event", event_release)
window.add(eventbox)

label = Gtk.Label("EventBox containing Label")
eventbox.add(label)

window.show_all()

Gtk.main()
```

Download: `EventBox`

CHAPTER

THIRTYSEVEN

BUTTONBOX

A ButtonBox is an invisible object which allows for manipulation and placement of Button widgets. This is commonly used in preference dialogs which have button widgets in a particular order with certain spacing requirements. There are two varieties of ButtonBox; horizontal and vertical.

37.1 Constructor

The ButtonBox can be constructed using the following:

```
buttonbox = Gtk.ButtonBox(orientation)
```

The *orientation* argument should be set to either `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`. However, the default for the ButtonBox is the horizontal mode.

37.2 Methods

Button widgets can be added to the ButtonBox using:

```
buttonbox.add(button)
```

They can also be removed if needed by calling:

```
buttonbox.remove(button)
```

ButtonBox supports a number of layouts to customise the appearance of your application. This is done by using:

```
buttonbox.set_layout(layout)
```

The *layout* parameter must be set to one of the following: `Gtk.ButtonBoxStyle.SPREAD` forces the buttons to spread out over the maximum space they have available. `Gtk.ButtonBoxStyle.EDGE` places the buttons at the edges of the ButtonBox. `Gtk.ButtonBoxStyle.START` and `Gtk.ButtonBoxStyle.END` place buttons at the start or end of the ButtonBox, depending on the container orientation in use. `Gtk.ButtonBoxStyle.CENTER` places buttons in the middle of the ButtonBox container.

By default, there is no spacing between each button with the ButtonBox. This can be configured with:

```
buttonbox.set_spacing(spacing)
```

In some cases it is useful to have a button within the ButtonBox be positioned seperately from the other buttons. This is commonly used for 'Help' or 'About' buttons, and can be set by doing:

```
buttonbox.set_child_secondary(child, is_secondary)
```

The *child* parameter is the name of the Button which is to be positioned separately. Setting the *is_secondary* parameter to `True` will separate the button.

By default, all buttons within the ButtonBox are homogeneous. It may be necessary to force one button to be non-homogeneous (for size reasons). This is achievable with:

```
buttonbox.set_child_non_homogeneous(child, non_homogeneous)
```

The *child* parameter is the name of the Button which is to be set as non-homogeneous. The *non_homogeneous* argument should be set to `True` to exempt the child from being homogeneous with the other child buttons.

37.3 Example

Below is an example of a ButtonBox:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ButtonBox(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        buttonbox = Gtk.ButtonBox()
        buttonbox.set_orientation(Gtk.Orientation.HORIZONTAL)
        buttonbox.set_spacing(2)
        self.add(buttonbox)

        button = Gtk.Button(label="Sparrow")
        buttonbox.add(button)
        button = Gtk.Button(label="Wren")
        buttonbox.add(button)
        button = Gtk.Button(label="Great Spotted Woodpecker")
        buttonbox.add(button)

window = ButtonBox()
window.show_all()

Gtk.main()
```

Download: `ButtonBox`

CHAPTER

THIRTYEIGHT

ADJUSTMENT

An Adjustment is an object which stores values relating to minimum, maximum and current values. It is invisible to the user and is used as a backend to other widgets such as the SpinButton or Scale.

This object is used in conjunction with a *SpinButton*, *Scale*, *ScaleButton*, or *Scrollbar*.

38.1 Constructor

The Adjustment can be constructed using the following:

```
adjustment = Gtk.Adjustment(value, lower, upper, step_increment, page_increment, page_
↪size)
```

The *value* is the initial number which the Adjustment starts with. The *lower* and *upper* values are the limits which the Adjustment allows. A *step_increment* is the standard number which the value is adjusted by whereas the *page_increment* value is the major number to adjust by. The *page_size* is only used by some widgets and determines the viewable area. In most cases, this will be set to `0` as it is not required.

38.2 Methods

To set the value of the Adjustment:

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

Retrieval of the value from the Adjustment is done with:

```
adjustment.get_value()
```

There are a range of functions which can be used to retrieve the configuration values from the Adjustment:

```
adjustment.get_lower()
adjustment.get_upper()
adjustment.get_step_increment()
adjustment.get_page_increment()
adjustment.get_page_size()
```

The configuration values can also be set after construction of the object with:

```
adjustment.set_lower(lower)
adjustment.set_upper(upper)
adjustment.set_step_increment(step_increment)
adjustment.set_page_increment(page_increment)
adjustment.set_page_size(page_size)
```

Adjustment values can also be set using a single method similar to the constructor with the call:

```
adjustment.configure(value, lower, upper, step_increment, page_increment, page_size)
```

The `.configure()` method will ensure that the `"changed"` signal is emitted once, rather than multiple times as is caused by the setting methods above.

38.3 Signals

The commonly used signals of an Adjustment are:

```
"changed" (adjustment)
"value-changed" (adjustment)
```

The `"changed"` signal emits from the Adjustment when any of the values are changed. Alternatively, the `"value-changed"` signal is emitted when just the value of the Adjustment has been modified. In both cases, the signals pass the Adjustment parameter.

38.4 Example

An example of an Adjustment in use can be seen with other widgets such as SpinButton, Scale, ScaleButton, or Scrollbar.

CHAPTER

THIRTYNINE

PROGRESSBAR

A ProgressBar is used to display progress of an operation visually. It is used when there is a long-running operation in progress, which requires the user to be updated on how long it will take or how much has been completed.

39.1 Constructor

The ProgressBar can be constructed using the following:

```
progressbar = Gtk.ProgressBar(orientation)
```

The *orientation* value allows the ProgressBar to be set to operate in two ways. By default, `Gtk.Orientation.HORIZONTAL` is used forcing the ProgressBar to move left-to-right. Alternatively, using `Gtk.Orientation.VERTICAL` allows the ProgressBar to move from top-to-bottom.

39.2 Methods

To set the amount of job which has been completed call:

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

The *fraction* parameter should be a value between `0.0` and `1.0`, with the float value indicating the percentage of the job that has been completed.

Alternatively, the amount completed can be retrieved using:

```
fraction = progressbar.get_fraction()
```

The value returned to the *fraction* variable will be a value between `0.0` and `1.0`.

To display text within the ProgressBar, use:

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

The *text* parameter should be a string of text informing the user of time remaining or number remaining.

If text is to be displayed using the `.set_text()` method, then you will also need to explicitly show the text with:

```
progressbar.set_show_text(show_text)
```

Setting the *show_text* argument to `True` displays whatever text was provided to the `.set_text()` method. If no text was set using the `.set_text()` method, then the percentage completion of the ProgressBar is used.

In some cases it may not be known how much of a job has been completed however it is still required to display feedback to the user. An alternative to setting a percentage completion is to use:

```
progressbar.pulse()
```

Using pulsing allows the bar to bounce from side-to-side indicating that a job is in progress. Every time the bar is to be moved, the `.pulse()` method should be called. It is commonly used within a loop which is executed while the job is in progress.

The step each pulse increments is set via:

```
progressbar.set_pulse_step(step)
```

The *step* value should be a number between `0.0` and `1.0`.

In some cases, particularly when using right-to-left languages, it may be useful to also set the ProgressBar to operate backwards. This can be set with:

```
progressbar.set_inverted(inverted)
```

When *inverted* is set to `True`, a horizontal bar will move from right-to-left and a vertical will move bottom-to-top.

39.3 Example

Below is an example of a ProgressBar:

```
#!/usr/bin/env python3

from gi.repository import Gtk, GObject

class ProgressBar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        self.progressbar = Gtk.ProgressBar()
        self.progressbar.set_show_text(True)
        self.add(self.progressbar)

        GObject.timeout_add(500, self.update_progressbar)

    def update_progressbar(self):
        fraction = self.progressbar.get_fraction() + 0.1

        if fraction <= 1.0:
            self.progressbar.set_fraction(fraction)
        else:
            self.progressbar.set_fraction(0.0)

        return True

window = ProgressBar()
window.show_all()

Gtk.main()
```

Download: `ProgressBar`

CHAPTER

FORTY

LEVELBAR

The LevelBar widget can be used as a level indicator. Typical usage cases would be displaying the strength of a password or the charge of a battery.

40.1 Constructor

The LevelBar is constructed with the following:

```
levelbar = Gtk.LevelBar()
```

Minimum and maximum values of the LevelBar can also be specified when the widget is constructed via:

```
levelbar = Gtk.LevelBar.new_for_interval(min_value, max_value)
```

The *min_value* and *max_value* should be integers which define the minimum and maximum permissable values the LevelBar allows.

40.2 Methods

A minimum and maximum value should be set on the LevelBar which defines the range of values which can be entered:

```
levelbar.set_min_value(value)
levelbar.set_max_value(value)
```

If required, the minimum and maximum values can also be retrieved:

```
levelbar.get_min_value()
levelbar.get_max_value()
```

The actual level can then be displayed:

```
levelbar.set_value(value)
```

To retrieve the value from the levelbar, use the method:

```
levelbar.get_value()
```

The update mode of the LevelBar indicates how the widget updates when the value is changed. There are two modes for this:

```
levelbar.set_mode(mode)
```

The *mode* value can be set to either `Gtk.LevelBarMode.CONTINUOUS`. This updates the LevelBar continuously with each update. Alternatively, `Gtk.LevelBarMode.DISCRETE` causes the LevelBar to update when the LevelBar stops value stops changing for a short time.

By default, horizontal LevelBar widgets fill from left-to-right, and vertical widgets update from bottom-to-top. This operation can be inverted however via:

```
levelbar.set_inverted(inverted)
```

A LevelBar provides an offset value which places a marker on the widget for a specified value.

levelbar.add_offset_value(name, value)

The *name* parameter is a textual name identifying the offset value. The *value* itself can be either an integer or decimal. When an offset value with the same name is provided, the existing value is overwritten.

To remove the offset value, call:

```
levelbar.remove_offset_value(name)
```

Finally, retrieval of the offset value is possible via:

```
levelbar.get_offset_value(name)
```

40.3 Signals

The commonly used signals of a LevelBar are:

```
"offset-changed" (name)
```

Whenever an offset value associated with the LevelBar is changed, the `"offset-changed"` is emitted, along with the name of the offset which has been adjusted.

40.4 Example

Below is an example of a LevelBar:

```
#!/usr/bin/env python3

from gi.repository import Gtk
import random

class LevelBar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(150, -1)
        self.connect("destroy", Gtk.main_quit)

        levelbar = Gtk.LevelBar()
        levelbar.set_min_value(0)
        levelbar.set_max_value(10)
```

(continues on next page)

(continued from previous page)

```
        levelbar.set_value(random.randint(0, 10))
        self.add(levelbar)

window = LevelBar()
window.show_all()

Gtk.main()
```

Download: `LevelBar`

CHAPTER

FORTYONE

NOTEBOOK

A Notebook widget is a container which displays content in a tab-based fashion. The differing content can be viewed by clicking on appropriate tabs.

41.1 Constructor

The Notebook can be constructed using the following:

```
notebook = Gtk.Notebook()
```

41.2 Methods

There are three options for adding pages to a Notebook. These are as follows:

```
notebook.append_page(child, tab_label)
notebook.prepend_page(child, tab_label)
notebook.insert_page(child, tab_label, position)
```

The `.append_page()` method adds pages to the end of the Notebook whereas `.prepend_page()` adds to the beginning. Calling `.insert_page()` adds pages to a particular location identified by *position*. The *child* parameter is the name of the widget which will be added to the Notebook page (commonly a Grid or Box). The *tab_label* specifies the Label widget which holds the title text of the page.

Pages can also be removed by specifying the position by calling:

```
notebook.remove_page(position)
```

The *position* value is a number identifying the current position of the tab to be removed, with `0` indicating the first tab.

To retrieve the number of pages which the Notebook holds:

```
n_pages = notebook.get_n_pages()
```

Retrieval of the child contained at a particular page number can be retrieved with:

```
nth_page = notebook.get_nth_page()
```

To return the page number of the currently selected page call:

```
position = notebook.get_current_page()
```

Setting the currently active page on the Notebook can be achieved with:

```
notebook.set_current_page(position)
```

By default, the Notebook shows tabs of all pages, however in some cases it may be useful to turn off the tab bar (for example if only one tab is visible). This can be toggled calling:

```
notebook.set_show_tabs(show_tabs)
```

If the Notebook contains a large number of tabs, it is recommended to enable scrolling to prevent the tab titles shrinking too much and preventing the title from being visible. This can be set with:

```
notebook.set_scrollable(scrollable)
```

Another useful feature would be to allow reordering of the tabs within the Notebook:

```
notebook.set_tab_reorderable(reorderable)
```

When *reorderable* is set to `True`, the user can drag-and-drop tabs into alternate positions.

In some cases, advanced functionality with extra widgets may need to be provided. This can be setup by using an Action Area:

```
notebook.set_action_widget(child, pack_type)
```

The *child* argument should be set to the widget which is to be added (usually this would be a Box). The *pack_type* parameter should be set to one of either `Gtk.PackType.START` or `Gtk.PackStart.END` which controls where the Action Area is positioned.

41.3 Signals

The common signals of the Notebook are:

```
"page-added" (notebook, child, page_num)
"page-removed" (notebook, child, page_num)
"page-reordered" (notebook, child, page_num)
"switch-page" (notebook, page, page_num)
```

The `"page-added"` and `"page-removed"` signals emit the *child* widget and the page number in *page_num* whenever a page is added or removed from the Notebook. The same values are also emitted whenever a Notebook page is moved via `"page-reordered"`. The `"switch-page"` signal emits each time the user changes which tab is active. This emits the values *page* which contains the page widget, and the *page_num* identifying the current page number index.

41.4 Example

Below is an example of a Notebook:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Notebook(Gtk.Window):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title('Notebook')
        self.set_default_size(300, 200)
        self.connect('destroy', Gtk.main_quit)

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

        for page in range(1, 4):
            label1 = Gtk.Label('Notebook')
            label2 = Gtk.Label()
            label2.set_text('Page %i' % (page))
            notebook.append_page(label1, label2)
            notebook.set_tab_reorderable(label1, True)

window = Notebook()
window.show_all()

Gtk.main()
```

Download: `Notebook`

CHAPTER

FORTYTWO

STACK

The Stack widget is a container which shows a single widget at a time. The widget is similar to a *Notebook* but it provides no means for the user to switch which child is visible.

If there is a requirement for the user to switch which child is visible, a *StackSwitcher* can be used.

42.1 Constructor

The Stack can be constructed using:

```
stack = Gtk.Stack()
```

42.2 Methods

To add child widgets to the Stack, use the method:

```
stack.add_named(child, name)
```

The *child* parameter is the name of the widget to be displayed in the Stack. The *name* specified is an identifier for the child.

If a StackSwitch is going to be used, a title can also be provided when adding the child widget.

    stack.add_title(child, name, title)

To set the visible child, use:

```
stack.set_visible_child(child)
stack.set_visible_child_name(name)
```

When using `.set_visible_child_name()`, the name is the string provided when adding the child to the Stack.

To ensure that the Stack requests the same size for all child widgets call:

```
stack.set_homogeneous(homogeneous)
```

If *homogeneous* is set to `False`, the Stack will resize to the same size as the child everytime a new child is displayed.

A number of transition types can be set when switching child widgets:

```
stack.set_transition_type(transition_type)
```

The *transition_type* should be set to one of:

- Gtk.StackTransitionType.NONE
- Gtk.StackTransitionType.CROSSFADE
- Gtk.StackTransitionType.SLIDE_RIGHT
- Gtk.StackTransitionType.SLIDE_LEFT
- Gtk.StackTransitionType.SLIDE_UP
- Gtk.StackTransitionType.SLIDE_DOWN
- Gtk.StackTransitionType.SLIDE_LEFT_RIGHT
- Gtk.StackTransitionType.SLIDE_UP_DOWN.

42.3 Example

Below is an example of an Stack:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Stack(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(400, 300)
        self.connect("destroy", Gtk.main_quit)

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

        self.stack = Gtk.Stack()
        self.stack.set_vexpand(True)
        self.stack.set_hexpand(True)
        grid.attach(self.stack, 0, 0, 1, 1)

        buttonbox = Gtk.ButtonBox()
        buttonbox.set_layout(Gtk.ButtonBoxStyle.CENTER)
        grid.attach(buttonbox, 0, 1, 1, 1)

        for page in range(1, 4):
            label = Gtk.Label("Stack Content on Page %i" % (page))
            name = "label%i" % page
            self.stack.add_named(label, name)

            button = Gtk.Button("Page %i" % (page))
            button.connect("clicked", self.on_button_clicked, page)
            buttonbox.add(button)

    def on_button_clicked(self, button, page):
        name = "label%i" % page
        self.stack.set_visible_child_name(name)

window = Stack()
window.show_all()

Gtk.main()
```

Download: `Stack`

CHAPTER

FORTYTHREE

STACKSWITCHER

The StackSwitcher is a controller for the *Stack* widget. It allows switching between the various child widgets of the Stack.

43.1 Constructor

The StackSwitcher is constructed using the call:

```
stackswitcher = Gtk.StackSwitcher()
```

43.2 Methods

To add the Stack which is to be controlled by the StackSwitcher, call:

```
stackswitcher.set_stack(stack)
```

Retrieval of the attached Stack is made with:

```
stack = stackswitcher.get_stack()
```

43.3 Example

Below is an example of an StackSwitcher:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class StackSwitcher(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(400, 300)
        self.connect("destroy", Gtk.main_quit)

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

(continues on next page)

(continued from previous page)

```
        stack = Gtk.Stack()
        stack.set_hexpand(True)
        stack.set_vexpand(True)
        grid.attach(stack, 0, 1, 1, 1)

        stackswitcher = Gtk.StackSwitcher()
        stackswitcher.set_stack(stack)
        grid.attach(stackswitcher, 0, 0, 1, 1)

        for page in range(1, 4):
            label = Gtk.Label("Stack Content on Page %i" % (page))
            name = "label%i" % page
            title = "Page %i" % page
            stack.add_titled(label, name, title)

window = StackSwitcher()
window.show_all()

Gtk.main()
```

Download: `StackSwitcher`

CHAPTER

FORTYFOUR

STACKSIDEBAR

The StackSidebar provides a way to switch between *Stack* objects via a list. The list is automatically populated with the children held by the Stack container.

Common use cases of the StackSidebar include preference screens, or other displays with many options which should be grouped together.

44.1 Constructor

The StackSidebar is constructed using:

```
stacksidebar = Gtk.StackSidebar()
```

44.2 Methods

To attach the Stack container to the StackSidebar, call:

```
stacksidebar.set_stack(stack)
```

The Stack associated with the StackSidebar can also be retrieved via:

```
stacksidebar.get_stack()
```

44.3 Property

The Stack object can also be handled by the properties:

```
stacksidebar.get_property("stack")
stacksidebar.set_property("stack", stack)
```

44.4 Example

Below is an example of an StackSidebar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class StackSidebar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(400, 300)
        self.connect("destroy", Gtk.main_quit)

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

        stack = Gtk.Stack()
        stack.set_hexpand(True)
        stack.set_vexpand(True)
        grid.attach(stack, 1, 0, 1, 1)

        stacksidebar = Gtk.StackSidebar()
        stacksidebar.set_stack(stack)
        grid.attach(stacksidebar, 0, 0, 1, 1)

        for page in range(1, 4):
            label = Gtk.Label("Stack Content on Page %i" % (page))
            name = "label%i" % page
            title = "Page %i" % page
            stack.add_titled(label, name, title)

window = StackSidebar()
window.show_all()

Gtk.main()
```

Download: `StackSidebar`

CHAPTER

FORTYFIVE

SCALE

A Scale widget is used to display the progress or amount on a sliding scale. The Scale can be adjusted to change the value as desired.

45.1 Constructor

A Scale can be constructed using the following:

```
scale = Gtk.Scale(orientation, adjustment)
```

The *orientation* parameter should be set to either `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`. The *adjustment* parameter should be set to specify the minimum, maximum and incremental values.

45.2 Methods

A Scale can be oriented in two directions; horizontal and vertical. To change the orientation after construction:

```
scale.set_orientation(orientation)
```

The *orientation* parameter should be set with the same values as those at construction.

By default, the Scale shows the value above the slider. This can be configured using:

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

When *draw_value* is set to False, the value is not displayed above the slider.

To change the position of where the value is displayed, use:

```
scale.set_value_pos(position)
```

The *position* value should be set to one of the following; `Gtk.PositionType.TOP`, `Gtk.PositionType.BOTTOM`, `Gtk.PositionType.LEFT` or `Gtk.PositionType.RIGHT`. By default `Gtk.PositionType.TOP` is used.

To adjust the number of decimal points displayed when drawing the value:

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

The *digits* parameter should be an integer figure, with 1 meaning 1.0, 2 meaning 1.00, etc. Setting the number of digits also causes the value to be rounded off.

To highlight the change between the initial value and the current value, the following method can be used to indicate the difference:

scale.set_has_origin(has_origin)

In some cases it may be useful to place a marker on the Scale to identify a particular position:

```
scale.add_mark(value, position, markup)
```

The *value* parameter should specify the value at which the mark is to be drawn. The *position* can also be specified, with one of the values previously mentioned accepted. The *markup* parameter can be set to a string of text which is also displayed on the scale next to the mark. This parameter is optional however.

All the marks can be cleared from the Scale with:

```
scale.clear_marks()
```

45.3 Signals

The common signals of a Scale widget are:

```
"format-value" (value)
```

The "format value" signal allows the way the value is displayed.

45.4 Example

Below is an example of a Scale:

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

from gi.repository import Gtk
import random

class Scale(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

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

        value = random.randint(0, 100)
        adjustment = Gtk.Adjustment(value, 0, 100, 1, 10, 0)

        self.scale = Gtk.Scale(orientation=Gtk.Orientation.VERTICAL,␣
↪adjustment=adjustment)
        self.scale.set_value_pos(Gtk.PositionType.BOTTOM)
        self.scale.set_vexpand(True)
        self.scale.set_hexpand(True)
        grid.attach(self.scale, 0, 0, 2, 1)

        buttonAdd = Gtk.Button(label="Add Mark")
```

(continues on next page)

(continued from previous page)

```
        buttonAdd.connect("clicked", self.on_add_mark_clicked)
        grid.attach(buttonAdd, 0, 1, 1, 1)

        buttonClear = Gtk.Button(label="Clear Marks")
        buttonClear.connect("clicked", self.on_clear_marks_clicked)
        grid.attach(buttonClear, 1, 1, 1, 1)

        radiobuttonVertical = Gtk.RadioButton(group=None, label="Vertical Scale")
        radiobuttonVertical.orientation = 0
        radiobuttonVertical.connect("toggled", self.on_orientation_clicked)
        grid.attach(radiobuttonVertical, 0, 3, 2, 1)

        radiobuttonHorizontal = Gtk.RadioButton(group=radiobuttonVertical, label=
↪"Horizontal Scale")
        radiobuttonHorizontal.orientation = 1
        radiobuttonHorizontal.connect("toggled", self.on_orientation_clicked)
        grid.attach(radiobuttonHorizontal, 0, 2, 2, 1)

    def on_add_mark_clicked(self, button):
        value = self.scale.get_value()
        self.scale.add_mark(value, Gtk.PositionType.LEFT, "Mark")

    def on_clear_marks_clicked(self, button):
        self.scale.clear_marks()

    def on_orientation_clicked(self, radiobutton):
        if radiobutton.orientation == 0:
            self.scale.set_orientation(Gtk.Orientation.VERTICAL)
        else:
            self.scale.set_orientation(Gtk.Orientation.HORIZONTAL)

window = Scale()
window.show_all()

Gtk.main()
```

Download: `Scale`

CHAPTER

FORTYSIX

SCROLLBAR

A Scrollbar provides the ability to navigate through content that is larger than the container it is positioned within.

In most cases a *ScrolledWindow* would be preferably as it automatically determines whether scrollbars are required, and the size they should be set to.

46.1 Constructor

The Scrollbar can be constructed using the following:

```
scrollbar = Gtk.Scrollbar(orientation, adjustment)
```

The *orientation* parameter indicates the direction of the Scrollbar and should be set to either `Gtk.Orientation.VERTICAL` or `Gtk.Orientation.HORIZONTAL`. An *adjustment* can also be specified which provides the values relating to the size of the content.

46.2 Example

Below is an example of a Scrollbar:

```
#!/usr/bin/env python3

from gi.repository import Gtk

window = Gtk.Window()
window.set_default_size(200, 200)
window.connect("destroy", Gtk.main_quit)

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

layout = Gtk.Layout()
layout.set_size(800, 500)
layout.set_vexpand(True)
layout.set_hexpand(True)

button = Gtk.Button(label="Button 1")
layout.put(button, 300, 400)
button = Gtk.Button(label="Button 2")
layout.put(button, 150, 50)
button = Gtk.Button(label="Button 3")
```

(continues on next page)

(continued from previous page)

```
layout.put(button, 720, 470)
grid.attach(layout, 0, 0, 1, 1)

vadjustment = layout.get_vadjustment()
hadjustment = layout.get_hadjustment()

vscrollbar = Gtk.Scrollbar(orientation=Gtk.Orientation.VERTICAL,␣
↪adjustment=vadjustment)
grid.attach(vscrollbar, 1, 0, 1, 1)
hscrollbar = Gtk.Scrollbar(orientation=Gtk.Orientation.HORIZONTAL,␣
↪adjustment=hadjustment)
grid.attach(hscrollbar, 0, 1, 1, 1)

window.show_all()

Gtk.main()
```

Download: `Scrollbar`

CHAPTER

FORTYSEVEN

SCROLLEDWINDOW

A ScrolledWindow provides scrolling functionality for widgets which are larger than their container. A ScrolledWindow automatically provides *Scrollbar* widgets automatically and would be used in most cases.

47.1 Constructor

The ScrolledWindow can be constructed using the following:

```
scrolledwindow = Gtk.ScrolledWindow(hadjustment, vadjustment)
```

The *hadjustment* and *vadjustment* parameters are optional, but can be used to specify the *Adjustment* objects for scrolling a widget.

47.2 Methods

Content can be added to the ScrolledWindow with:

```
scrolledwindow.add(child)
scrolledwindow.add_with_viewport(child)
```

The `.add()` method must be used with widgets that support scrolling such as TextView, TreeView or IconView. Alternatively, `.add_with_viewport()` should be used for widgets that do not natively allow scrolling such as Box or Grid.

Child widgets can be removed if needed by calling:

```
scrolledwindow.remove(child)
```

By default, the scrollbars are both displayed regardless of whether they are active or not. To configure this policy use:

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

Both parameters can be setting to one of the following; `Gtk.PolicyType.ALWAYS`, `Gtk.PolicyType.AUTOMATIC` or `Gtk.PolicyType.NEVER`. When `Gtk.PolicyType.ALWAYS` is used, the scrollbars are always shown. `Gtk.PolicyType.NEVER` sets the scrollbars to never show, even if the content is larger than the container. The `Gtk.PolicyType.AUTOMATIC` constant sets the scrollbar to display if the content is larger than the container, otherwise it is not shown.

The *hadjustment* and *vadjustment* methods can be added after construction using:

```
scrolledwindow.set_hadjustment(adjustment)
scrolledwindow.set_vadjustment(adjustment)
```

The default setting is that the scrollbars should be positioned on the right and along the bottom. In most cases this should be sufficient, however to change the position of the scrollbar objects use:

```
scrolledwindow.set_placement(window_placement)
```

The *window_placement* argument should be configured to one of the following; `Gtk.CornerType.TOPLEFT`, `Gtk.CornerType.TOPRIGHT`, `Gtk.CornerType.BOTTOMLEFT` or `Gtk.CornerType.BOTTOMRIGHT`.

The placement can be unset if required:

```
scrolledwindow.unset_placement()
```

To prevent content shrinking to a size which is deemed too small, call:

```
scrolledwindow.set_min_content_height(height)
scrolledwindow.set_min_content_width(width)
```

The *width* value should be specified in pixels.

47.3 Example

Below is an example of an ScrolledWindow:

```
#!/usr/bin/env python3

from gi.repository import Gtk

window = Gtk.Window()
window.set_default_size(200, 200)
window.connect("destroy", Gtk.main_quit)

scrolledwindow = Gtk.ScrolledWindow()
window.add(scrolledwindow)

layout = Gtk.Layout()
layout.set_size(800, 600)
layout.set_vexpand(True)
layout.set_hexpand(True)
scrolledwindow.add(layout)

hadjustment = layout.get_hadjustment()
scrolledwindow.set_hadjustment(hadjustment)
vadjustment = layout.get_vadjustment()
scrolledwindow.set_vadjustment(vadjustment)

button = Gtk.Button(label="Button 1")
layout.put(button, 645, 140)
button = Gtk.Button(label="Button 2")
layout.put(button, 130, 225)
button = Gtk.Button(label="Button 3")
layout.put(button, 680, 350)
```

(continues on next page)

(continued from previous page)

```
window.show_all()

Gtk.main()
```

Download: `ScrolledWindow`

CHAPTER

FORTYEIGHT

VIEWPORT

A Viewport adds the ability to scroll widgets which do not natively support scrolling (e.g. Grid, Box).

48.1 Constructor

The Viewport can be constructed using the following:

```
viewport = Gtk.Viewport(hadjustment, vadjustment)
```

The *hadjustment* and *vadjustment* parameters are optional and can be specified *Adjustment* objects.

48.2 Methods

Rather than create Adjustment objects manually, these can be retrieved from the Viewport with:

```
hadjustment = viewport.get_hadjustment()
vadjustment = viewport.get_vadjustment()
```

If you do use manually created Adjustment objects, these can be attached after construction by calling:

```
viewport.set_hadjustment(hadjustment)
viewport.set_vadjustment(vadjustment)
```

The shadow type places a shadow type around the Viewport, and is set via:

```
viewport.set_shadow_type(shadow)
```

The *shadow* should be set to one of the following:

- `Gtk.ShadowType.NONE` - no outline.
- `Gtk.ShadowType.IN` - outline bevelled inwards.
- `Gtk.ShadowType.OUT` - outline bevelled outwards.
- `Gtk.ShadowType.ETCHED_IN` - outline sunken.
- `Gtk.ShadowType.ERCHED_OUT` - outline raised.

48.3 Example

Below is an example of a Viewport:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Viewport(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(-1, 200)
        self.connect("destroy", Gtk.main_quit)

        scrolledwindow = Gtk.ScrolledWindow()
        self.add(scrolledwindow)

        hadjustment = Gtk.Adjustment()
        vadjustment = Gtk.Adjustment()

        viewport = Gtk.Viewport(hadjustment, vadjustment)
        scrolledwindow.add(viewport)

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        viewport.add(box)

        for i in range(1, 16):
            button = Gtk.Button(label="Button %s" % i)
            box.pack_start(button, True, True, 0)

window = Viewport()
window.show_all()

Gtk.main()
```

Download: `Viewport`

CHAPTER

FORTYNINE

FRAME

A Frame is a container widget used to group other widgets which perform related functions.

49.1 Constructor

The Frame can be constructed using the following:

```
frame = Gtk.Frame(label)
```

The *label* parameter should be set to a string of text which is displayed on the Frame to indicate what the group of child widgets relates to.

49.2 Methods

To set the label text on the frame after constructing use:

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

Alternatively, a *Label* widget can be used on the Frame if required by calling:

```
frame.set_label_widget(label_widget)
```

By default, the label is positioned in the top-left corner of the Frame. This can be changed using:

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

The *xalign* and *yalign* values should be float values between `0.0` and `1.0`. By default, these values are set to 0.0 and 0.5 on a newly created Frame.

49.3 Example

Below is an example of a Frame:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Frame(Gtk.Window):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.set_border_width(5)
        self.connect("destroy", Gtk.main_quit)

        frame = Gtk.Frame(label="Frame")
        self.add(frame)

        label = Gtk.Label("Label in a Frame")
        frame.add(label)

window = Frame()
window.show_all()

Gtk.main()
```

Download: `Frame`

CHAPTER

FIFTY

ASPECTFRAME

A AspectFrame is a container widget used to group other widgets which perform related functions. It differs from a *Frame* in that child content is scaled depending on the child size, or the specified settngs of the container.

An AspectFrame is based on a Frame and therefore provides many of the same methods.

50.1 Constructor

The AspectFrame can be constructed using the following:

```
aspectframe = Gtk.AspectFrame(label, xalign, yalign, ratio, obey_child)
```

The *label* parameter should be set to a string of text which is displayed on the AspectFrame to indicate what the group of child widgets relates to. The *xalign* and *yalign* parameters specify the position of the child within the AspectFrame. The *ratio* value indicates the size of the child with relation to its initial size, and can be set to a float value. Finally, the *obey_child* takes the ratio of the child widget and uses that to determine the AspectFrame size when set to `True`.

Note: The *ratio* value only applies when *obey_child* is set to `False`.

50.2 Methods

To set the label text on the frame after constructing use:

```
aspectframe.set_label(label)
```

Alternatively, a *Label* widget can be used on the AspectFrame if required by calling:

```
aspectframe.set_label_widget(label_widget)
```

By default, the label is positioned in the top-left corner of the AspectFrame. This can be changed using:

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

The *xalign* and *yalign* values should be float values between `0.0` and `1.0`. By default, these values are set to 0.0 and 0.5 on a newly created AspectFrame.

To set the AspectFrame properties relating to the child size after construction use:

```
aspectframe.set(xalign, yalign, ratio, obey_child)
```

50.3 Example

Below is an example of a AspectFrame:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class AspectFrame(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.set_border_width(5)
        self.connect("destroy", Gtk.main_quit)

        frame = Gtk.AspectFrame(label="AspectFrame",
                                xalign=0.5,
                                yalign=0.5,
                                ratio=1,
                                obey_child=False)
        self.add(frame)

        label = Gtk.Label("Label in an AspectFrame")
        frame.add(label)

window = AspectFrame()
window.show_all()

Gtk.main()
```

Download: `AspectFrame`

CHAPTER

FIFTYONE

EXPANDER

An Expander allows seldom used content to be hidden until the user requests it to be displayed. This can be useful for configuration settings which should be changed rarely.

51.1 Constructor

The Expander can be constructed using the following:

```
expander = Gtk.Expander(label)
```

The *label* parameter must be set to a textual string which identifies what the Expander is or contains.

51.2 Methods

Items can be added to the Expander widget with:

```
expander.add(child)
```

The *child* can be set to any widget type, however it is typical for a Grid or Box to be displayed and other widgets such as Button or Label objects to be added on top of the container.

It can also later be removed via:

```
expander.remove(child)
```

Once a child has been added, it may be necessary add spacing between the top of the Expander and the child content:

```
expander.set_spacing(spacing)
```

The *spacing* value should be an integer identifying the number of pixels worth of space to insert.

To set the Expander label after creation call:

```
expander.set_label(label)
```

A *Label* widget can also be applied if necessary via the method:

```
expander.set_label_widget(widget)
```

To set the Expander to open programatically the following can be called:

```
expander.set_expanded(expanded)
```

When set to `True`, the Expander will be opened and the child content will be displayed.

Expanding the Expander widget will cause the parent widget (i.e. Window, Dialog) to expand to make room for the content within the Expander. The default action when closing the Expander is to leave the Window at the largest size, rather than shrink to the original. To change this behaviour use:

```
expander.set_resize_toplevel(resize)
```

When *resize* is set to `True`, the parent will shrink once the Expander is collapsed.

51.3 Signals

The commonly used signals of an Expander are:

```
"activate" (expander)
```

The `"activate"` signal emits from the Expander when the content is shown or hidden.

51.4 Example

Below is an example of a Expander:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Expander(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, -1)
        self.connect("destroy", Gtk.main_quit)

        expander = Gtk.Expander(label="Expander")
        expander.set_resize_toplevel(True)
        self.add(expander)

        label = Gtk.Label("Label in an Expander")
        label.set_size_request(200, 200)
        expander.add(label)

window = Expander()
window.show_all()

Gtk.main()
```

Download: `Expander`

CHAPTER

FIFTYTWO

REVEALER

The Revealer widget behaves similar to an *Expander*, however does this using an animation. The child widget can be hidden or unhidden based on the action of another widget such as a button.

52.1 Constructor

The Revealer is constructed using the following:

```
revealer = Gtk.Revealer()
```

52.2 Methods

Child widgets are added to the Revealer using:

```
revealer.add(child)
```

The item can also be removed from the container via:

```
revealer.remove(child)
```

By default, the Revealer is collapsed. To configure whether it is expanded or collapsed use:

```
revealer.set_reveal_child(revealed)
```

If *revealed* is set to `True`, the Revealer expands to reveal the child widget. When `False` is specified, the Revealer is collapsed again.

A number of transition types can be used to animate the Revealer using the method:

```
revealer.set_transition_type(transition_type)
```

The *transition_type* can be set to one of the following:

- `Gtk.TransitionType.NONE`
- `Gtk.TransitionType.CROSSFADE`
- `Gtk.TransitionType.SLIDE_RIGHT`
- `Gtk.TransitionType.SLIDE_LEFT`
- `Gtk.TransitionType.SLIDE_UP`

- `Gtk.TransitionType.SLIDE_DOWN`

The duration of the animation can also be configured:

```
revealer.set_transition_duration(duration)
```

The *duration* value is a value in milliseconds. By default, the Revealer takes 250 milliseconds to complete the transition.

52.3 Example

Below is an example of a Revealer:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Revealer(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

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

        self.revealer = Gtk.Revealer()
        self.revealer.set_reveal_child(True)
        grid.attach(self.revealer, 0, 0, 1, 1)

        label = Gtk.Label("Label in a Revealer")
        self.revealer.add(label)

        button = Gtk.Button("Reveal")
        button.connect("clicked", self.on_reveal_clicked)
        grid.attach(button, 0, 1, 1, 1)

    def on_reveal_clicked(self, button):
        reveal = self.revealer.get_reveal_child()
        self.revealer.set_reveal_child(not reveal)

window = Revealer()
window.show_all()

Gtk.main()
```

Download: `Revealer`

CHAPTER

FIFTYTHREE

SIZEGROUP

The SizeGroup object providess a mechanism for grouping widgets, and that all widgets within the group equal the same size regardless of content. It is similar in some ways to the *ButtonBox* widget.

53.1 Constructor

The SizeGroup can be constructed using the following:

```
sizegroup = Gtk.SizeGroup(mode)
```

The *mode* constant determines how the SizeGroup displays the widgets and should be set to:

- `Gtk.SizeGroup.NONE` - do not size equally.
- `Gtk.SizeGroup.HORIZONTAL` - size equally horizontally.
- `Gtk.SizeGroup.VERTICAL` - size equally vertically.
- `Gtk.SizeGroup.BOTH` - size equally both horizontally and vertically.

53.2 Methods

Widgets are added to and removed from the SizeGroup using:

```
sizegroup.add_widget(widget)
sizegroup.remove_widget(widget)
```

To retrieve a list of widgets which are currently within the SizeGroup use:

```
sizegroup.get_widgets()
```

The SizeGroup mode can also be set after construction with:

```
sizegroup.set_mode(mode)
```

The *mode* parameter should be set with one of the constants defined in the construction section.

To configure whether hidden widgets which are a member of the SizeGroup are taken into account call:

```
sizegroup.set_ignore_hidden(ignore_hidden)
```

When *ignore_hidden* is set to `True`, the size allocation for the widgets will be re-calculated to account for the hidden widget.

53.3 Example

Below is an example of a SizeGroup:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class SizeGroup(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        box = Gtk.Box(orientation=Gtk.Orientation.HORIZONTAL, spacing=2)
        self.add(box)

        sizegroup = Gtk.SizeGroup()
        sizegroup.set_mode(Gtk.SizeGroupMode.HORIZONTAL)

        button = Gtk.Button(label="Arch")
        sizegroup.add_widget(button)
        box.pack_start(button, True, True, 0)

        button = Gtk.Button(label="Debian")
        sizegroup.add_widget(button)
        box.pack_start(button, True, True, 0)

        button = Gtk.Button(label="Red Hat\nEnterprise Linux")
        sizegroup.add_widget(button)
        box.pack_start(button, True, True, 0)

window = SizeGroup()
window.show_all()

Gtk.main()
```

Download: `SizeGroup`

CHAPTER

FIFTYFOUR

PANED

A Paned widget provides an adjustable split-screen view. It is commonly used in email clients to separate the various interface elements.

54.1 Constructor

The Paned can be constructed using the following:

```
paned = Gtk.Paned(orientation)
```

The *orientation* argument should be set to either `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`.

54.2 Methods

To add items to the Paned widget, there are two methods:

```
paned.add1(child)
paned.add2(child)
```

The `.add1()` method adds the child widget to the top or left pane of the container. Calling `.add2()` places the child in the bottom or right pane. The child in most cases will be a Grid or Box.

If more flexible packing functionality is requires, the following can be run:

```
paned.pack1(child, resize, shrink)
paned.pack2(child, resize, shrink)
```

The *resize* argument determines whether the child should expand when the Paned is resized. Setting this to `True` set the widget to expand. Setting the *shrink* argument to `True` also allows the child to be made smaller than its available area.

To set the position of the expander, call:

```
paned.set_position(position)
```

The *position* value should be an integer in pixels which specifies how wide the left or top pane is.

Retrieving the position from the Paned widget is done via the method:

```
position = paned.get_position()
```

The *position* value returned is the number of pixels wide the left or top pane is. This is useful when saving the application configuration to a file.

The Paned object also supports a wide-handle, which makes adjustments to the Paned sizes easier:

```
paned.set_wide_handle(wide_handle)
```

When *wide_handle* is set to `True`, the wide-handle view is enabled.

54.3 Signals

The commonly used signals of a Paned are:

```
"move-handle" (widget, scroll_type)
```

The `"move-handle"` signals emits two properties when it occurs. The *widget* is that of the Paned upon which the handle move occurred. The *scroll_type* returns the type of scroll action.

54.4 Example

Below is an example of a Paned:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Paned(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title('Paned')
        self.set_default_size(400, 200)
        self.connect('destroy', Gtk.main_quit)

        hpaned = Gtk.Paned()
        hpaned.set_position(150)
        self.add(hpaned)

        label = Gtk.Label(label='Left Pane')
        hpaned.add1(label)

        vpaned = Gtk.Paned(orientation=Gtk.Orientation.VERTICAL)
        vpaned.set_position(75)
        hpaned.add2(vpaned)

        label = Gtk.Label(label='Top Right Pane')
        vpaned.add1(label)

        label = Gtk.Label(label='Bottom Right Pane')
        vpaned.add2(label)

window = Paned()
window.show_all()

Gtk.main()
```

Download: `Paned`

CHAPTER

FIFTYFIVE

FIXED

A Fixed container provides a surface which allows the positioning of child widgets at fixed co-ordinates.

Fixed is very similar to the :doc‘layout‘ container, however it doesn’t provide an infinite scrolling area. It should be used when the size of the Fixed area will be known.

55.1 Constructor

The Fixed can be constructed using the following:

```
fixed = Gtk.Fixed()
```

55.2 Methods

To position widgets on the Fixed use:

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

The *widget* value should be the child widget which is being added to the Fixed container. Specifying the *x* and *y* values specifies the co-ordinates that the child should be placed at.

To move the child widget after construction call:

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

55.3 Example

Below is an example of a Fixed:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Fixed(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)
```

(continues on next page)

(continued from previous page)

```
        fixed = Gtk.Fixed()
        self.add(fixed)

        button = Gtk.Button(label="Button 1")
        fixed.put(button, 40, 60)
        button = Gtk.Button(label="Button 2")
        fixed.put(button, 120, 95)

window = Fixed()
window.show_all()

Gtk.main()
```

Download: `Fixed`

CHAPTER

FIFTYSIX

LAYOUT

A Layout provides an infinite scrolling area for child widgets to be placed on. It can also be used as a surface for custom drawings to be displayed.

If the size of the scrolling area is going to be known before running, it is suggested that a *Fixed* is used.

56.1 Constructor

The Layout can be constructed using the following:

```
layout = Gtk.Layout(hadjustment, vadjustment)
```

The *hadjustment* and *vadjustment* parameters should be set to an *Adjustment* object which stores the values of the scrollbars.

56.2 Methods

To position widgets on the Layout use:

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

The *widget* parameter specifies the widget which is to be drawn on the Layout. The *x* and *y* values are the co-ordinates of where to place the widget.

Widgets can be moved after calling `.put()` with:

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

To set the size of the Layout call:

```
layout.set_size(width, height)
```

Alternatively, to retrieve the Layout size use:

```
size = layout.get_size()
```

The Adjustment objects can be specified after constructing the Layout with:

```
layout.set_hadjustment(adjustment)
layout.set_vadjustment(adjustment)
```

56.3 Example

Below is an example of a Layout:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Layout(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        layout = Gtk.Layout()
        self.add(layout)

        button = Gtk.Button(label="Button 1")
        layout.put(button, 40, 60)
        button = Gtk.Button(label="Button 2")
        layout.put(button, 120, 95)

window = Layout()
window.show_all()

Gtk.main()
```

Download: `Layout`

CHAPTER

FIFTYSEVEN

OVERLAY

An Overlay widget provides a container which a child widget can be packed into. The widget is displayed over a larger, background widget to provide a floating object.

57.1 Constructor

The Overlay can be constructed using the following:

```
overlay = Gtk.Overlay()
```

57.2 Methods

The background or larger item can be added to the overlay with the method:

```
overlay.add(widget)
```

Overlay widgets can be then added using:

```
overlay.add_overlay(widget)
```

Input made on the Overlay widget can be passed through to the unlying widget with:

```
overlay.set_overlay_pass_through(widget, pass_through)
```

The *widget* parameter should be set to the underlying widget which will receive input. The *pass_through* parameter should be set to `True` or `False` as to whether this functionality is enabled or not.

57.3 Example

Below is an example of a Overlay:

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

from gi.repository import Gtk

window = Gtk.Window()
window.set_default_size(200, 200)
window.connect("destroy", Gtk.main_quit)
```

(continues on next page)

(continued from previous page)

```
overlay = Gtk.Overlay()
window.add(overlay)

textview = Gtk.TextView()
textview.set_wrap_mode(Gtk.WrapMode.WORD_CHAR)
textbuffer = textview.get_buffer()
textbuffer.set_text("Welcome to the PyGObject Tutorial\n\nThis guide aims to provide␣
↪an introduction to using Python and GTK+.\n\nIt includes many sample code files and␣
↪exercises for building your knowledge of the language.", -1)
overlay.add(textview)

button = Gtk.Button(label="Overlayed Button")
button.set_valign(Gtk.Align.CENTER)
button.set_halign(Gtk.Align.CENTER)
overlay.add_overlay(button)

overlay.show_all()

window.show_all()

Gtk.main()
```

Download: `Overlay`

CHAPTER

FIFTYEIGHT

MENUBAR

A MenuBar provides the basis of a application menu, with access to menus such as File, Edit, Help, etc being displayed within the MenuBar.

58.1 Constructor

The MenuBar can be constructed using the following:

```
menubar = Gtk.MenuBar()
```

58.2 Methods

Items can be added to the MenuBar after construction with:

```
menubar.append(menuitem)
menubar.prepend(menuitem)
menubar.insert(menuitem, position)
```

The `.append()` method packs items as they are called. The `.prepend()` method adds items to the front of the MenuBar in the order they are called. Using the `.insert()` method allows for the insertion of MenuItem widgets to a particular location within the MenuBar.

By default, items are packed to the left of the MenuBar. This can be changed via the method:

```
menubar.set_pack_direction(pack_direction)
```

The *pack_direction* can be set to `Gtk.PackDirection.RTL` which pushes items to the right, `Gtk.PackDirection.TTB` which stacks widgets beneath the previous one, or `Gtk.PackDirection.BTT` which stacks widgets on top of one another. The default value is `Gtk.PackDirection.LTR`.

58.3 Example

For an example of the MenuBar see the *Menu* page.

CHAPTER

FIFTYNINE

MENU

There are two types of menu which can be constructed. Both use the Menu widget however are used for different implementations. These are the Application menu which is displayed via a *MenuBar*, and provides access to all features of the application. Alternatively, a context menu appears when the user right-clicks on an interface and provides access to common functions.

59.1 Constructor

The Menu can be constructed using the following:

```
menu = Gtk.Menu()
```

59.2 Methods

Items can be added to the Menu using the following three methods:

```
menu.append(menuitem)
menu.prepend(menuitem)
menu.insert(menuitem, position)
```

The *menuitem* parameter should be the MenuItem which is to be added to the Menu. Items added using the `.append()` method are added at the end of the Menu, whereas when using the `.prepend()` method items are added to the top of the Menu. When using the `.insert()` parameter, a *position* must be specified which indicates the position where the items are to be added within the Menu.

It is also possible to remove items from the Menu with:

```
menu.remove(menuitem)
```

To move an item within a Menu call:

```
menu.reorder_child(menuitem, position)
```

An *AccelGroup* can be attached to the Menu using:

```
menu.set_accel_group(accelgroup)
```

Enabling the Menu to reserve space for menu items as it would for toggles or icons, this can be enabled via:

```
menu.set_reserve_toggle_size(reserve_toggle)
```

When the *reserve_toggle* value is set to `True`, the text will be indented within the menu regardless of whether there are any icons or toggles.

59.3 Example

Below is an example of a Menu, MenuBar and associated MenuItem widgets:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def event(widget, event):
    if event.button == 3:
        menu.popup(None, None, None, None, event.button, event.time)
        menu.show_all()

    return True

window = Gtk.Window()
window.connect("destroy", Gtk.main_quit)

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

menubar = Gtk.MenuBar()
menubar.set_hexpand(True)
grid.attach(menubar, 0, 0, 1, 1)

menuitem = Gtk.MenuItem(label="MenuItem")
menubar.append(menuitem)
menu = Gtk.Menu()
menuitem.set_submenu(menu)
menuitem = Gtk.MenuItem(label="MenuItem")
menu.append(menuitem)

menuitem = Gtk.MenuItem(label="CheckMenuItem")
menubar.append(menuitem)
menu = Gtk.Menu()
menuitem.set_submenu(menu)
checkmenuitem = Gtk.CheckMenuItem(label="CheckMenuItem")
menu.append(checkmenuitem)

menuitem = Gtk.MenuItem(label="RadioMenuItem")
menubar.append(menuitem)
menu = Gtk.Menu()
menuitem.set_submenu(menu)
radiomenuitem1 = Gtk.RadioMenuItem(label="RadioMenuItem 1")
radiomenuitem1.set_active(True)
menu.append(radiomenuitem1)
radiomenuitem2 = Gtk.RadioMenuItem(label="RadioMenuItem 2", group=radiomenuitem1)
menu.append(radiomenuitem2)

eventbox = Gtk.EventBox()
```

(continues on next page)

(continued from previous page)

```
eventbox.set_size_request(-1, 200)
eventbox.connect("button-release-event", event)
grid.attach(eventbox, 0, 1, 1, 1)

window.show_all()

Gtk.main()
```

Download: `Menu`

CHAPTER

SIXTY

MENUITEM

A MenuItem widget is the most basic of the menu item family, and allows a string of text to be displayed.

60.1 Constructor

menuitem = Gtk.MenuItem(label, use_underline)

The *label* parameter defines the text which is to be shown on the MenuItem. The *use_underline* parameter should be set to `True` to identify the character with an preceding underscore as an mnemonic key.

60.2 Methods

Setting the label text after construction is done using:

```
menuitem.set_label(label)
```

To set whether the MenuItem uses a mnemonic key after constructing call:

```
menuitem.set_use_underline(use_underline)
```

Submenus can be attached to a MenuItem via:

```
menuitem.set_submenu(menu)
```

The *menu* value should be a named *Menu* widget. This should be used for both menus attached to items on a Menubar, and when constructing a tree of menus.

A MenuItem can be programatically activated with the method:

```
menuitem.activate()
```

60.3 Signals

The common signals of a MenuItem are:

```
"activate" (menuitem)
"activate-item" (menuitem)
```

The "activate" signal emits when the user clicks on the MenuItem. The "activate-item" is emitted when the user clicks a MenuItem as with "activate", however it also emits when a submenu appears. This would be useful for dynamic loading of items in a submenu. In most cases, "activate" is the correct signal to use.

60.4 Example

For an example of the MenuItem see the *Menu* page.

CHAPTER

SIXTYONE

CHECKMENUITEM

The CheckMenuItem provides a *CheckButton* for use within a menu structure. It displays a box with associated label that can indicate a checked or unchecked state.

61.1 Constructor

The CheckMenuItem can be constructed using the following:

```
checkmenuitem = Gtk.CheckMenuItem(label, use_underline)
```

61.2 Methods

The label can be applied to the CheckMenuItem after constructing it with:

```
checkmenitem.set_label(label)
```

To retrieve the state of the CheckMenuItem use:

```
checkmenuitem.get_active()
```

The state can also be set with:

```
checkmenuitem.set_active(active)
```

When *active* is set to `True`, the CheckMenuItem displays a tick in the box indicating an active state.

When using multiple CheckMenuItem widgets, it may be necessary to indicate an inconsistent state if all other items in the group have a `True` or `False` value. This can be set via:

```
checkmenuitem.set_inconsistent(inconsistent)
```

To retrieve whether a CheckMenuItem is in an inconsitent state use:

```
checkmenuitem.get_inconsistent()
```

61.3 Signals

The common signals of the CheckMenuItem are:

```
"toggled" (checkmenuitem)
```

The "toggled" signal emits from the widget when the user changes the state to checked or unchecked.

61.4 Example

For an example of the MenuItem see the *Menu* page.

CHAPTER

SIXTYTWO

RADIOMENUITEM

A RadioMenuItem is the menu-based equivalent of a *RadioButton*, but is also very similar to a *CheckMenuItem* with the exception that it can be added to a group. With the group specified, only one RadioMenuItem in the group can be active at any one time.

62.1 Constructor

The RadioMenuItem can be constructed using the following:

```
radiomenuitem = Gtk.RadioMenuItem(label, group, use_underline)
```

The *label* parameter should be set to a string of text which identifies the function of the RadioMenuItem. A *group* must also be declared to allow the RadioMenuItem to function. For the first RadioMenuItem in the group, this would be set to `None`, with all subsequent widgets being added to the group being the name of the first widget added. The *use_underline* value can also be set as `True` to provide a mnemonic key which provides accessibility via the keyboard.

62.2 Methods

The label text of the RadioMenuItem can be specified after creation with:

```
radiomenuitem.set_label(label)
```

A group can also be defined after construction of the widget:

```
radiomenuitem.join_group(group)
```

The *group* parameter should be set to the first RadioMenuItem which is to be a part of the group. The first item however does not need to have the `.join_group()` method called.

To retrieve the group which the RadioMenuItem is attached:

```
radiomenuitem.get_group()
```

62.3 Signals

The commonly used signals of the RadioMenuItem are:

```
"toggled" (radiomenuitem)
"group-changed" (radiomenuitem)
```

A "toggled" signal is emitted by the RadioMenuItem whenever the widget is made active or inactive. The "group-changed" signal is emitted when the RadioMenuItem group is changed, or it is removed from the group entirely.

62.4 Example

For an example of the RadioMenuItem see the *Menu* page.

CHAPTER

SIXTYTHREE

SEPARATORMENUITEM

A SeparatorMenuItem is very similar to a *Separator* however is designed for use within a *Menu* widget. The widget displays a horizontal line with shadow to give the appearance of an indentation, and its use is for logical separation of items within the menu.

63.1 Constructor

The SeparatorMenuItem can be constructed using the following:

```
separatormenuitem = Gtk.SeparatorMenuItem()
```

63.2 Example

For an example of the SeparatorMenuItem see the *Menu* page.

CHAPTER

SIXTYFOUR

EXPANDER

The MenuButton widget displays a menu when the button is clicked.

64.1 Constructor

The MenuButton can be constructed using the following:

```
menubutton = Gtk.MenuButton()
```

64.2 Methods

A MenuButton can have a *Menu* added to it:

```
menubutton.set_popup(menu)
```

By default, the menu appears beneath the MenuButton, however this can be configured:

```
menubutton.set_direction(direction)
```

The *direction* parameter can be set to one of the following:

- `Gtk.ArrowType.NONE`
- `Gtk.ArrowType.DOWN`
- `Gtk.ArrowType.UP`
- `Gtk.ArrowType.LEFT`
- `Gtk.ArrowType.RIGHT`

64.3 Example

Below is an example of a MenuButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class MenuButton(Gtk.Window):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        menubutton = Gtk.MenuButton("MenuButton")
        self.add(menubutton)

        menu = Gtk.Menu()
        menubutton.set_popup(menu)

        for count in range(1, 6):
            menuitem = Gtk.MenuItem("Item %i" % (count))
            menuitem.connect("activate", self.on_menuitem_activated)
            menu.append(menuitem)

        menu.show_all()

    def on_menuitem_activated(self, menuitem):
        print("%s Activated" % (menuitem.get_label()))

window = MenuButton()
window.show_all()

Gtk.main()
```

Download: `MenuButton`

CHAPTER

SIXTYFIVE

POPOVER

A Popover widget provides a bubble-like context window which is attached to a widget. It is also commonly used for menus in place of a *Menu*.

65.1 Constructor

The Popover is constructed by calling:

```
popover = Gtk.Popover()
```

Alternatively, the Popover can be attached to the widget which will launch it by:

```
popover = Gtk.Popover(relative_to)
```

When the Popover is to be used as a replacement for a Menu widget, it can be built with a MenuModel listing the menu items:

```
popover = Gtk.Popover.new_from_model(model)
```

65.2 Methods

Items are added to the Popover with:

```
popover.add(child)
```

The *child* parameter should be set to name of the widget to be added to the Popover. Typically this will be a container such as a *Box* or *Grid* from which further widgets can be added.

The Popover is attached to the widget:

```
popover.set_relative_to(relative_to)
```

The *relative_to* parameter should be set to the widget the Popover is to be attached to.

The position of the Popover menu can be set using the method:

```
popover.set_position(position)
```

The *position* value can be set to one of the following:

- `Gtk.PositionType.TOP`

- `Gtk.PositionType.BOTTOM`
- `Gtk.PositionType.LEFT`
- `Gtk.PositionType.RIGHT`

In some cases, it may be required to ensure that when the Popover is displayed, it is the focus of the input until closed. This is set with:

```
popover.set_modal(modal)
```

When *modal* is set to `True`, the Popover will have the keyboard focus.

The default widget within the Popover can be set using the method:

```
popover.set_default_widget(widget)
```

65.3 Example

Below is an example of a Popover:

```
#!/usr/bin/env python3

from gi.repository import Gtk


def button_clicked(button):
    popover.show_all()

window = Gtk.Window()
window.set_default_size(250, 250)
window.connect("destroy", Gtk.main_quit)

box = Gtk.Box()
box.set_orientation(Gtk.Orientation.VERTICAL)
window.add(box)

button = Gtk.Button("Popover Launcher")
button.connect("clicked", button_clicked)
box.add(button)

popover = Gtk.Popover()
popover.set_position(Gtk.PositionType.RIGHT)
popover.set_relative_to(button)

box = Gtk.Box()
box.set_spacing(5)
box.set_orientation(Gtk.Orientation.VERTICAL)
popover.add(box)

label = Gtk.Label("A Label widget")
box.add(label)

checkbutton = Gtk.CheckButton("A CheckButton widget")
box.add(checkbutton)

window.show_all()
```

(continues on next page)

(continued from previous page)

```
Gtk.main()
```

Download: `Popover`

CHAPTER

SIXTYSIX

TOOLBAR

The Toolbar widget provides access to common features of an application such as Save, Print or Find.

66.1 Constructor

A Toolbar can be constructed using the following:

```
toolbar = Gtk.Toolbar()
```

66.2 Methods

Items can be added to the Toolbar with:

```
toolbar.insert(item, position)
```

The *item* should be an appropriate ToolItem object. A *position* value indicates the location the item is to be added on the Toolbar, with 0 designating the first place on the Toolbar.

Items can also be removed using:

```
toolbar.remove(position)
```

To retrieve the number of items which are currently displayed on the Toolbar run:

```
n_items = toolbar.get_n_items()
```

Alternatively, retrieving the widget at a specified position use:

```
child = toolbar.get_nth_item()
```

By default, the Toolbar style is specified in the GTK+ global setting. This can be configured with:

```
toolpalette.set_style(style)
```

The *style* parameter should be set to one of the following; `Gtk.ToolbarStyle.ICONS`, `Gtk.ToolbarStyle.TEXT`, `Gtk.ToolbarStyle.BOTH`, `Gtk.ToolbarStyle.BOTH_HORIZ`. When using `Gtk.ToolbarStyle.ICONS` or `Gtk.ToolbarStyle.TEXT` only icons or text are displayed. `Gtk.ToolbarStyle.BOTH` displays icons and text, with the text positioned beneath the icon. `Gtk.ToolbarStyle.BOTH_HORIZ` displays text to the left of the icon.

Note: It is highly recommended not to set a style. This allows GTK+ to use the global setting which ensures your application is consistent with others.

Icon sizes within the Toolbar can be configured with:

```
toolbar.set_icon_size(icon_size)
```

The *icon_size* argument must be set to one of the following values; `Gtk.IconSize.INVALID`, `Gtk.IconSize.MENU`, `Gtk.IconSize.SMALL_TOOLBAR`, `Gtk.IconSize.LARGE_TOOLBAR`, `Gtk.IconSize.BUTTON`, `Gtk.IconSize.DND`, or `Gtk.IconSize.DIALOG`. The Toolbar takes the global size setting by default.

If the number of icons is larger than the space allocated for the Toolbar, a menu can be shown to provide access to those which have overflown. This functionality can be configured with:

```
toolbar.set_show_arrow(show_arrow)
```

When *show_arrow* is set to `True`, the menu will be shown when clicking on a button which displays the remaining items in a menu.

66.3 Example

Below is an example of a Toolbar and associated items that can be attached:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Toolbar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(500, -1)
        self.connect("destroy", Gtk.main_quit)

        toolbar = Gtk.Toolbar()
        self.add(toolbar)

        toolbutton = Gtk.ToolButton()
        toolbutton.set_label("New")
        toolbutton.set_is_important(True)
        toolbutton.set_icon_name("gtk-new")
        toolbar.add(toolbutton)

        toggletoolbutton = Gtk.ToggleToolButton()
        toggletoolbutton.set_icon_name("gtk-media-play")
        toolbar.add(toggletoolbutton)

        menu = Gtk.Menu()

        menuitem = Gtk.MenuItem(label="MenuItem")
        menu.append(menuitem)
        menuitem.show()

        menutoolbutton = Gtk.MenuToolButton()
        menutoolbutton.set_menu(menu)
```

(continues on next page)

(continued from previous page)

```
        menutoolbutton.set_icon_name("gtk-open")
        toolbar.add(menutoolbutton)

        separatortoolitem = Gtk.SeparatorToolItem()
        toolbar.add(separatortoolitem)

        radiotoolbutton1 = Gtk.RadioToolButton()
        radiotoolbutton1.set_icon_name("gtk-media-rewind")
        toolbar.add(radiotoolbutton1)
        radiotoolbutton2 = Gtk.RadioToolButton(group=radiotoolbutton1)
        radiotoolbutton2.set_icon_name("gtk-media-forward")
        toolbar.add(radiotoolbutton2)

        separatortoolitem = Gtk.SeparatorToolItem()
        toolbar.add(separatortoolitem)

        toolitem = Gtk.ToolItem()
        toolbar.add(toolitem)
        entry = Gtk.Entry()
        toolitem.add(entry)

window = Toolbar()
window.show_all()

Gtk.main()
```

Download: `Toolbar`

CHAPTER

SIXTYSEVEN

TOOLPALETTE

A ToolPalette is used to display a large number of items. It is similar to a *Toolbar*, however is tailored for use in applications which require more features accessible to the user such as image editing.

67.1 Constructor

The ToolPalette can be constructed using the following:

```
toolpalette = Gtk.ToolPalette()
```

67.2 Methods

In some cases, it may be required to only display one of the ToolItemGroup objects at any one time with:

```
toolpalette.set_exclusive(toolitemgroup, exclusive)
```

The *toolitemgroup* parameter should be set to a :doc:`toolitemgroup` identifying which should be set to exclusive status. When the *exclusive* parameter is set to ``True``, the ToolItemGroup will display and all other open groups will be closed.

To expand a ToolItemGroup programatically use:

```
toolpalette.set_expand(toolitemgroup, expand)
```

Again, the *toolitemgroup* parameter is a ToolItemGroup which is to be expanded. When the *expand* property is set to `True`, the items within the group will be opened.

Groups can be set or changed with:

```
toolpalette.set_group_position(group, position)
```

The *group* parameter must be set to a ToolItemGroup which is to be configured. The *position* value should be an integer value indicating the location of the group within the ToolPalette, with `0` indicating the first position.

The ToolPalette by default takes the system style for Toolbars. This can be configured to:

```
toolpalette.set_style(style)
```

The *style* parameter should be set to one of the following; `Gtk.ToolbarStyle.ICONS`, `Gtk.ToolbarStyle.TEXT`, `Gtk.ToolbarStyle.BOTH`, `Gtk.ToolbarStyle.BOTH_HORIZ`. When using

`Gtk.ToolbarStyle.ICONS` or `Gtk.ToolbarStyle.TEXT` only icons or text are displayed. `Gtk.ToolbarStyle.BOTH` displays icons and text, with the text positioned beneath the icon. `Gtk.ToolbarStyle.BOTH_HORIZ` displays text to the left of the icon.

Note: It is highly recommended not to set a style and to use whichever the GTK+ global setting specifies, for both consistency and usability reasons.

Icon sizes within the ToolPalette can be configured with:

```
toolpalette.set_icon_size(icon_size)
```

The *icon_size* argument must be set to one of the following values; `Gtk.IconSize.INVALID`, `Gtk.IconSize.MENU`, `Gtk.IconSize.SMALL_TOOLBAR`, `Gtk.IconSize.LARGE_TOOLBAR`, `Gtk.IconSize.BUTTON`, `Gtk.IconSize.DND`, or `Gtk.IconSize.DIALOG`. The ToolPalette takes the global size setting by default.

Customisation of the direction in which items are displayed in the ToolPalette can be made with:

```
toolpalette.set_orientation(orientation)
```

The *orientation* parameter by default is `Gtk.Orientation.VERTICAL` in which all groups of added vertically, however `Gtk.Orientation.HORIZONTAL` can also be used to add groups horizontally.

67.3 Example

Below is an example of a ToolPalette:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ToolPalette(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        toolpalette = Gtk.ToolPalette()
        self.add(toolpalette)

        toolitemgroup = Gtk.ToolItemGroup(label="Group 1")
        toolpalette.add(toolitemgroup)

        toolbutton = Gtk.ToolButton()
        toolbutton.set_icon_name("gtk-new")
        toolitemgroup.insert(toolbutton, 0)
        toolbutton = Gtk.ToolButton()
        toolbutton.set_icon_name("gtk-open")
        toolitemgroup.insert(toolbutton, 1)
        toolbutton = Gtk.ToolButton()
        toolbutton.set_icon_name("gtk-save")
        toolitemgroup.insert(toolbutton, 2)

        toolitemgroup = Gtk.ToolItemGroup(label="Group 2")
        toolpalette.add(toolitemgroup)
```

(continues on next page)

(continued from previous page)

```
        toolbutton = Gtk.ToolButton()
        toolbutton.set_icon_name("gtk-about")
        toolitemgroup.insert(toolbutton, 0)
        toolbutton = Gtk.ToolButton()
        toolbutton.set_icon_name("gtk-preferences")
        toolitemgroup.insert(toolbutton, 1)

window = ToolPalette()
window.show_all()

Gtk.main()
```

Download: `ToolPalette`

CHAPTER

SIXTYEIGHT

TOOLITEMGROUP

A ToolItemGroup is used to contain icons with a similar purpose within a *ToolPalette*. The group provides a title which identifies the child items, and also allows the group to be expanded or collapsed based on user-preference.

68.1 Constructor

The ToolItemGroup can be constructed using the following:

```
toolitemgroup = Gtk.ToolItemGroup(label)
```

Setting the *label* to a string of text attaches it to the top of the group.

68.2 Methods

To set the title of the group which names the group, use:

```
toolitemgroup.set_label(label)
```

Items should be added to the ToolItemGroup with the method:

```
toolitemgroup.insert(toolitem, position)
```

The *toolitem* parameter is to be set to the name of a ToolItem to be added to the group. The *position* parameter should be set to the positional value where the item is inserted to, with `0` indicating the first position.

A ToolItem position can also be changed with:

```
toolitemgroup.set_item_position(toolitem, position)
```

Alternatively, items can be removed from the group using:

```
toolitemgroup.remove(toolitem)
```

By default, all ToolItemGroup objects are set as expanded and all icons visible. However, it may be necessary in some cases to collapse some groups programatically:

```
toolitemgroup.set_collapsed(collapsed)
```

68.3 Example

For an example of the ToolItemGroup, see the *ToolPalette* page.

CHAPTER

SIXTYNINE

TOOLITEM

A ToolItem is the base of all other Toolbar-related widgets. It provides a blank item, from which standard widgets (i.e. *Entry*) can be added to the Toolbar.

The methods contained within a ToolItem can also be used for all other ToolItem-related widgets.

69.1 Constructor

The ToolItem can be constructed using:

```
toolitem = Gtk.ToolItem()
```

69.2 Methods

Widgets can be added and removed from ToolItem via:

```
toolitem.add(widget)
toolitem.remove(widget)
```

To expand a ToolItem to fill all empty space on the Toolbar use:

```
toolitem.set_expand(expand)
```

In some cases it may be necessary to show or hide the widget based on the orientation of the Toolbar via:

```
toolitem.set_visible_vertical(visible_vertical)
toolitem.set_visible_horizontal(visible_horizontal)
```

By default, the widget will take its size based only on its own content. To ensure that it stays the same size as other widgets in the Toolbar call:

```
toolitem.set_homogeneous(homogeneous)
```

69.3 Example

To view an example for this widget, see the *Toolbar* example.

CHAPTER

SEVENTY

TOOLBUTTON

A ToolButton is the most basic ToolItem. It should be used within a *Toolbar* or *ToolPalette* to provide clickable buttons. Each ToolButton can contain a label and/or image however the user settings may override the settings defined by the application.

70.1 Constructor

The ToolButton can be constructed using the following:

```
toolbutton = Gtk.ToolButton(label, icon_widget, icon_name)
```

The *label* parameter defines the string of text to be displayed on the ToolButton. In all circumstances this should be set. The *icon_widget* allows a widget such as an *Image* to be added, however can be omitted if not required. The *icon_name* allows the name of an icon to be used, which is then loaded from the current theme. This may also be omitted if not needed.

70.2 Methods

Custom label text can be used by specifying a string of text in:

```
toolbutton.set_label(label)
```

It is recommended to display a mnemonic character which is identified by using an underscore before the character which is to be identied as mnemonic. This can be turned on with:

```
toolbutton.set_use_underline(use_underline)
```

When *use_underline* is set to `True`, the underscore if removed and the later after displays a underline.

Alternatively, if custom *Label* or *Image* widgets need to be attached to a ToolButton, use:

```
toolbutton.set_label_widget(label_widget)
toolbutton.set_icon_widget(icon_widget)
```

An icon name can also be defined which allows an icon from the current theme to be loaded:

```
toolbutton.set_icon_name(icon_name)
```

By default, when the toolbar style is set to display text beside icons, no icons will display text. This is to ensure that only important, or frequently used icons display text, and are therefore more visible to users. This can be set with:

```
toolbutton.set_is_important(is_important)
```

When *is_important* is set to `True`, the icon text will be displayed.

Note: Icon text is always displayed when text is set to show below icons. This method only affects the text beside icons functionality.

Note: It is good interface design to ensure only important items in the Toolbar have the `.set_is_important()` method set.

In some use cases, it may be useful to have an item set to invisible if the Toolbar is configured to either horizontal or vertical mode:

```
toolbutton.set_visible_horizontal(visible_horizontal)
toolbutton.set_visible_vertical(visible_vertical)
```

By default, all items are shown whether vertical or horizontal. Setting either method to `False` will result in the item being hidden.

It is highly recommended to set a *Tooltip* on the ToolButton using:

```
toolbutton.set_tooltip_text(tooltip_text)
toolbutton.set_tooltip_markup(tooltip_markup)
```

70.3 Signals

The commonly used signals of an ToolButton are:

```
"clicked" (toolbutton)
```

The `"clicked"` signal emits from the ToolButton when the user presses and then releases the mouse button. It can also occur when the item has the focus and the user presses the `Return` button for example.

70.4 Example

To view an example for this widget, see the *Toolbar* example.

CHAPTER

SEVENTYONE

TOGGLETOOLBUTTON

A ToggleToolButton is similar to a *ToggleButton* in functionality however it should be applied to a *Toolbar*.

71.1 Constructor

The ToggleToolButton can be constructed using the following:

```
toggletoolbutton = Gtk.ToggleToolButton(label)
```

71.2 Methods

Note: The methods listed below only apply to this widget and those that inherit from it. For more methods, see the *ToolButton* page. For more information on widget hierarchy, see *Hierarchy Theory*.

To flip the active or inactive state, use the method:

```
toggletoolbutton.set_active(active)
```

When *active* is set to `True`, the ToggleToolButton will appear depressed.

To retrieve the current state of the ToggleToolButton call:

```
toggletoolbutton.get_active()
```

71.3 Signals

The commonly used signals of an ToggleToolButton are:

```
"toggled" (toggletoolbutton)
```

When the user clicks on the ToggleToolButton and the state is changed to active or inactive, the `"toggled"` signal is emitted.

71.4 Example

To view an example for this widget, see the *Toolbar* example.

CHAPTER

SEVENTYTWO

RADIOTOOLBUTTON

A RadioToolButton provides a widget similar to a *RadioButton* for use within a *Toolbar*. When used with other RadioToolButton widgets, only one in the group can be active at a single time.

72.1 Constructor

The RadioToolButton can be constructed using the following:

```
radiotoolbutton = Gtk.RadioToolButton(label, group)
```

The first constructor allows creation of a RadioToolButton with custom text. This is specified via the *label* paramter. The constructor also uses the *group* parameter identifiying the group the RadioToolButton belongs.

72.2 Methods

Note: The methods listed below only apply to this widget and those that inherit from it. For more methods, see the *ToolButton* page. For more information on widget hierarchy, see *Hierarchy Theory*.

A group can be applied to the RadioToolButton via:

```
radiotoolbutton.set_group(group)
```

The name of the group which the RadioToolButton is attached to can also be specified with:

```
radiotoolbutton.get_group()
```

An active RadioToolButton can be set programatically using:

```
radiotoolbutton.set_active(active)
```

If *active* is set to `True`, the RadioToolButton will appear checked, while others in the group will be unchecked.

To check whether a RadioToolButton is active use the method:

```
radiotoolbutton.get_active()
```

If the method returns `True`, the RadioToolButton in the group is currently active.

72.3 Signals

The commonly used signals of an RadioToolButton are:

```
"toggled" (radiotoolbutton)
```

When the user clicks on the RadioToolButton and the state is changed to active or inactive, the `"toggled"` signal is emitted.

72.4 Example

To view an example for this widget, see the *Toolbar* example.

CHAPTER

SEVENTYTHREE

MENUTOOLBUTTON

A MenuToolButton provides two functions; a standard *ToolButton* which can be clicked, and a *Menu* which provides more options relating to the button.

73.1 Constructor

The MenuToolButton is constructed with the call:

```
menutoolbutton = Gtk.MenuToolButton(label, icon_widget, icon_name)
```

The *label* parameter defines the string of text to be displayed on the MenuToolButton. In all circumstances this should be set. The *icon_widget* allows a widget such as an *Image* to be added, however can be omitted if not required. The *icon_name* allows the name of an icon to be used, which is then loaded from the current theme. This may also be omitted if not needed.

73.2 Methods

Note: The methods listed below only apply to this widget and those that inherit from it. For more methods, see the *ToolButton* page. For more information on widget hierarchy, see *Hierarchy Theory*.

A Menu can be attached after constructing with:

```
menutoolbutton.set_menu(menu)
```

The *menu* parameter supplied should be the name of a Menu object. If no Menu object is supplied, the arrow button will be set as insensitive.

The Menu object associated with the MenuToolButton can be retrieved if required with:

```
menutoolbutton.get_menu()
```

Specific text can be set as a tooltip for the menu button via:

```
menutoolbutton.set_arrow_tooltip_text(tooltip_text)
menutoolbutton.set_arrow_tooltip_markup(tooltip_markup)
```

For unformatted text, it is recommended to use the `.set_arrow_tooltip_text()` method. Enhanced, formatted text can be provided via `.set_arrow_tooltip_markup()`.

73.3 Signals

The commonly used signals for the widget are:

```
"clicked" (menutoolbutton)
"show-menu" (menutoolbutton)
```

Use of the "clicked" signal is made when the button portion of the MenuToolButton is clicked. The "show-menu" signal emits from the widget when the dropdown arrow is clicked, but before the actual menu is displayed, allowing for on-demand loading of menu items.

73.4 Example

To view an example for this widget, see the *Toolbar* example.

CHAPTER

SEVENTYFOUR

SEPARATORTOOLITEM

A SeparatorToolItem provides a *Separator* but is tailored for use with a *Toolbar*. It draws an indented line on the widget to allow for visual separation of items.

74.1 Constructor

A SeparatorToolItem can be constructed using the following:

```
separatortoolitem = Gtk.SeparatorToolItem()
```

74.2 Methods

To enable whether the widget is drawn use:

```
separatortoolitem.set_draw(draw)
```

When *draw* is set to `False`, the line is removed. This is useful for items which are frequently removed from the Toolbar, and leaving the line drawn would look poor. The default value is `True`.

74.3 Example

To view an example for this widget, see the *Toolbar* example.

CHAPTER

SEVENTYFIVE

PLACESSIDEBAR

The PlacesSidebar offers a list of frequently used locations in the file system. It is commonly found in file managers and also seen in the *FileChooser* family of widgets.

75.1 Constructor

The PlacesSidebar can be created with:

```
placessidebar = Gtk.PlacesSidebar()
```

75.2 Methods

The PlacesSidebar offers a number of flag options to configure how the widget provides links. The use of these flags is also dependant on what the application supports:

```
placessidebar.set_open_flags(flags)
```

The *flags* parameter can be set to one of the following; `Gtk.Places.OPEN_NORMAL`, `Gtk.Places.OPEN_NEW_TAB`, or `Gtk.Places.OPEN_NEW_WINDOW`. The default option is `Gtk.Places.OPEN_NORMAL` and tells the application to open the link in usual way.

To set whether the shortcut to the Desktop is shown use:

```
placessidebar.set_show_desktop(show_desktop)
```

When *show_desktop* is set to `True`, the shortcut will be shown. The default however is for this to be hidden.

An option to configure whether the Connect to Server option is shown can be made with:

```
placessidebar.set_show_connect_to_server(show_server)
```

If *show_server* is set to `False`, the option is hidden. This is useful if the application implements a way to connect to servers itself in a different way.

The desktop environment is able to determine whether recent items are displayed. This can be overridden via:

```
placessiderbar.set_show_recent(show_recent)
```

When *show_recent* is set to `False`, the recent items are not displayed.

To configure whether the trash location is visible, call:

```
placessidebar.set_show_trash(show_trash)
```

75.3 Signals

The common signals of the PlacesSidebar are:

```
"populate-popup" (placessidebar, menu, selected_item, selected_volume)
```

The `"populate-popup"` signal emits when the user invokes the context menu of the PlacesSidebar. A *menu* parameter emits containing the *Menu* widget, which allows appending of custom menu items. The *selected_item* points to the file item, or `None` if the item is a volume. The *selected-volume* value points to the volume passed, or `None` if the item is a file.

75.4 Example

Below is an example of a PlacesSidebar:

```
#!/usr/bin/env python3

from gi.repository import Gtk
from gi.repository import Gio

class PlacesSidebar(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        placessidebar = Gtk.PlacesSidebar()
        placessidebar.connect("open-location", self.on_open_location)
        self.add(placessidebar)

    def on_open_location(self, placessidebar, location, flags):
        location = placessidebar.get_location()

        print("Opened URI: %s" % (GLocalFile.get_uri(location)))

window = PlacesSidebar()
window.show_all()

Gtk.main()
```

Download: `PlacesSidebar`

CHAPTER

SEVENTYSIX

FILECHOOSER

The FileChooser is an interface which is used by *FileChooserButton*, *FileChooserWidget*, and *FileChooserDialog* and is constructed when those objects are constructed.

76.1 Methods

The default action of the FileChooser is to provide functionality to open files.

filechooser.set_action(action)

The *action* value can be set to one of the following; `Gtk.FileChooserAction.OPEN`, `Gtk.FileChooserAction.SAVE`, `Gtk.FileChooserAction.SELECT_FOLDER`, or `Gtk.FileChooserAction.CREATE_FOLDER`.

By default, the FileChooser allows selection of a single file only. This can be configured with:

```
filechooser.set_select_multiple(select_multiple)
```

The *select_multiple* can be set to `True` which allows the user to hold down `Control` and select the items with the mouse.

Retreival of the selected filename or uniform resource identifier (URI) is done via:

```
filename = widget.get_filename()
uri = widget.get_uri()
```

Alternatively, if you have provided the ability for multiple files to be selected, you must use:

```
filenames = widget.get_filenames()
uris = widget.get_uris()
```

Another useful function, particular when saving files is to predefine a filename for the file, for example "Unsaved Document".

filename.set_filanme(filename)

To configure whether the button to create new folders is visible, call:

```
widget.set_create_folders(create_folders)
```

The *create_folders* option should be set to `False` if the button is to be hidden.

76.2 Signals

The common signals of the FileChooser are as follows:

```
"file-activated" (filechooser)
```

The "file-activated" signal emits when the user double-clicks a file, or selects a file and presses Enter.

76.3 Example

To view an example for this widget, see the *FileChooserWidget*, :doc‘filechooserdialog‘, or *FileChooserButton* examples.

CHAPTER

SEVENTYSEVEN

FILECHOOSERWIDGET

A FileChooserWidget provides a widget for selecting files and folders from disks and network shares.

77.1 Constructor

The FileChooserWidget can be constructed using the following:

```
filechooserwidget = Gtk.FileChooserWidget(action)
```

The *action* parameter dictates what the FileChooserWidget can do. It should be set to one of `Gtk.FileChooserAction.OPEN` which allows opening of files, `Gtk.FileChooserAction.SAVE` which allows files to be saved, `Gtk.FileChooserAction.SELECT_FOLDER` which enables selecting of folders and `Gtk.FileChooserAction.CREATE_FOLDER` which creates folders based on a specified name.

77.2 Example

Below is an example of a FileChooserWidget:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def file_selected(button):
    print("Selected file: %s" % filechooserwidget.get_filename())
    Gtk.main_quit()

window = Gtk.Window()
window.set_default_size(600, 400)
window.connect("destroy", Gtk.main_quit)

box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL, spacing=5)
window.add(box)

filechooserwidget = Gtk.FileChooserWidget(action=Gtk.FileChooserAction.OPEN)
box.pack_start(filechooserwidget, True, True, 0)

buttonbox = Gtk.ButtonBox(orientation=Gtk.Orientation.HORIZONTAL)
buttonbox.set_layout(Gtk.ButtonBoxStyle.END)
buttonbox.set_spacing(2)
box.pack_start(buttonbox, False, False, 0)
```

(continues on next page)

(continued from previous page)

```
button = Gtk.Button(label="_Open", use_underline=True)
button.connect("clicked", file_selected)
buttonbox.add(button)

window.show_all()

Gtk.main()
```

Download: `FileChooserWidget`

CHAPTER

SEVENTYEIGHT

FILECHOOSERDIALOG

A FileChooserDialog provides a FileChooserWidget within a dialog window.

The dialog-variant of the FileChooser allows for selecting of files and folders.

78.1 Constructor

The FileChooserDialog can be constructed using the following:

```
filechooserdialog = Gtk.FileChooserDialog(action, buttons)
```

The *action* parameter should be set to either `Gtk.FileChooserAction.OPEN` which allows opening of files, `Gtk.FileChooserAction.SAVE` which allows files to be saved, `Gtk.FileChooserAction.SELECT_FOLDER` which enables selecting of folders and `Gtk.FileChooserAction.CREATE_FOLDER` which creates folders based on a specified name. The *buttons* pareameter is a tuple of buttons and response_id values which will be displayed in the dialog.

78.2 Methods

Once the FileChooserDialog has been constructed use:

```
filechooserdialog.run()
filechooserdialog.destroy()
```

Note: If your application only uses a FileChooserDialog, the `Gtk.main()` call is not required. This is invoked automatically when calling `filechooserdialog.run()`.

The title of the dialog which is displayed can be set with:

```
filechooserdialog.set_title(title)
```

When using a FileChooserDialog, the parent window should be defined. This ensures correct positioning of the dialog when it appears:

```
filechooserdialog.set_parent(parent)
```

Generally, the *parent* parameter will be a *Window* or some other dialog type.

78.3 Example

Below is an example of a FileChooserDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

filechooserdialog = Gtk.FileChooserDialog()
filechooserdialog.set_title("FileChooserDialog")
filechooserdialog.add_button("_Open", Gtk.ResponseType.OK)
filechooserdialog.add_button("_Cancel", Gtk.ResponseType.CANCEL)
filechooserdialog.set_default_response(Gtk.ResponseType.OK)

response = filechooserdialog.run()

if response == Gtk.ResponseType.OK:
    print("File selected: %s" % filechooserdialog.get_filename())

filechooserdialog.destroy()
```

Download: `FileChooserDialog`

CHAPTER

SEVENTYNINE

FILECHOOSERBUTTON

The FileChooserButton provides access to a *FileChooserDialog*. The button is used when space is limited to allow selecting of a file.

79.1 Constructor

The FileChooserButton can be constructed using the following:

```
filechooserbutton = Gtk.FileChooserButton(action)
```

The *action* parameter should be set to either `Gtk.FileChooserAction.OPEN` which allows opening of files, `Gtk.FileChooserAction.SAVE` which allows files to be saved, `Gtk.FileChooserAction.SELECT_FOLDER` which enables selecting of folders and `Gtk.FileChooserAction.CREATE_FOLDER` which creates folders based on a specified name.

79.2 Methods

To specify a title on the dialog that appears, use the method:

```
filechooserbutton.set_title(title)
```

79.3 Signals

The signals used by the FileChooserButton are:

```
"file-set" (filechooserbutton)
```

When the user picks a file, the `"file-set"` signal emits.

79.4 Example

Below is an example of a FileChooserButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def file_changed(filechooserbutton):
    print("File selected: %s" % filechooserbutton.get_filename())

window = Gtk.Window()
window.set_default_size(150, -1)
window.connect("destroy", Gtk.main_quit)

filechooserbutton = Gtk.FileChooserButton(title="FileChooserButton")
filechooserbutton.connect("file-set", file_changed)
window.add(filechooserbutton)

window.show_all()

Gtk.main()
```

Download: `FileChooserButton`

CHAPTER

EIGHTY

FILEFILTER

A FileFilter object is used in conjunction with FileChooser-based widgets, to limit the type of files which can be viewed. For example, to limit a FileChooser to only view music files, a FileFilter can be used.

80.1 Constructor

The FileFilter can be constructed using the following:

```
filefilter = Gtk.FileFilter()
```

Note: A FileFilter must be created for each file type which is to be filtered.

80.2 Methods

To set the name of the FileFilter use:

```
filefilter.set_name(name)
```

Retrieving the name of a particular filter is possible using:

```
filefilter.get_name()
```

To limit files which can be displayed by extension use:

```
filefilter.add_pattern(pattern)
```

The *pattern* parameter should be a string, which uses wildcard entries. For example using `"*.flac"` would permit only files which end with the .flac extension.

Files can also be limited according to their mime-type:

```
filefilter.add_mime_type(mime_type)
```

A *mime_type* is a content filter which uses content within the header of the file to identify it. Example of a mime types are `"image/png"`, `"video/mp4"`, and `"text/html"`.

80.3 Example

Below is an example of a FileFilter:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class FileFilter(Gtk.FileChooserDialog):
    def __init__(self):
        Gtk.FileChooserDialog.__init__(self)
        self.add_button("_Cancel", Gtk.ResponseType.CLOSE)
        self.add_button("_Open", Gtk.ResponseType.OK)
        self.connect("response", self.on_response)

        filefilter = Gtk.FileFilter()
        filefilter.set_name("All Items")
        filefilter.add_pattern("*")
        self.add_filter(filefilter)

        filefilter = Gtk.FileFilter()
        filefilter.set_name("Audio")
        filefilter.add_mime_type("audio/flac")
        filefilter.add_mime_type("audio/ogg")
        self.add_filter(filefilter)

        filefilter = Gtk.FileFilter()
        filefilter.set_name("Images")
        filefilter.add_pattern("*.png")
        filefilter.add_pattern("*.jpg")
        filefilter.add_pattern("*.bmp")
        self.add_filter(filefilter)

    def on_response(self, filechooserdialog, response):
        filechooserdialog.destroy()

dialog = FileFilter()
dialog.run()
```

Download: `FileFilter`

CHAPTER

EIGHTYONE

FONTCHOOSER

The FontChooser is an interface which is used by *FontChooserWidget*, *FontChooserDialog*, and *FontButton*.

81.1 Methods

To retrieve the font family, attributes (bold, italic, etc), or the size selected, use the methods:

```
fontchooser.get_font_family()
fontchooser.get_font_face()
fontchooser.get_font_size()
```

Finally, the call for retrieving the actual font selected is:

```
fontchooser.get_font()
```

To retrieve a normalised string from the FontChooser, describing the font selected use:

```
fontchooser.get_font_desc()
```

The `.get_font_desc()` method will return a string such as "Cantarell Bold Italic 12".

It is also possible to set the font using a description:

```
fontchooser.set_font_desc(font_desc)
```

ColorChooser objects also support setting a font:

```
fontchooser.set_font(fontname)
```

A preview entry can be used to allow the user to test the font selected. This can be enabled or disabled via:

```
fontchooser.set_show_preview_entry(show_entry)
```

The default value is `True` which shows the preview entry.

The preview text is the string which showcases the font, size, and other attributes. By default, this is set to "The quick brown fox jumped over the lazy dog.", however can be customised using:

```
fontchooser.set_preview_text(preview_text)
```

If required, the preview text can also be returned:

```
preview_text = fontchooser.get_preview_text()
```

81.2 Signals

The commonly used signals of an FontChooserDialog are:

```
"font-activated" (fontchooserdialog, font)
```

The `"font-activated"` event emits from the FontChooserDialog when a font is selected either via double-clicking with the mouse, or by pressing the Enter/Return key.

81.3 Example

To view an example of the FontChooser, see the code for the objects *FontChooserWidget*, *FontChooserDialog*, and *FontButton*.

CHAPTER

EIGHTYTWO

FONTCHOOSERWIDGET

A FontChooserWidget allows for the selection of fonts including the styling and size.

82.1 Constructor

The FontChooserWidget can be constructed using the following:

```
fontchooserwidget = Gtk.FontChooserWidget()
```

82.2 Example

Below is an example of a FontChooserWidget:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def font_chooser(fontchooserwidget, font):
    print("Font selected: %s" % font)

window = Gtk.Window()
window.connect("destroy", Gtk.main_quit)

fontchooserwidget = Gtk.FontChooserWidget()
fontchooserwidget.connect("font-activated", font_chooser)
window.add(fontchooserwidget)

window.show_all()

Gtk.main()
```

Download: `FontChooserWidget`

CHAPTER

EIGHTYTHREE

FONTCHOOSERDIALOG

The FontChooserDialog provides a *FontChooserWidget* within a dialog window.

The dialog-variant of the FontChooser family allows for choosing fonts, styling and sizes.

83.1 Constructor

The FontChooserDialog can be constructed using the following:

```
fontchooserdialog = Gtk.FontChooserDialog(title, parent)
```

A *title* should be specified via a string of text which identifies the function of the FontChooserDialog. The *parent* parameter can be the name of a Window or Dialog which owns the FontChooserDialog.

83.2 Methods

Once the FontChooserDialog has been constructed use:

```
fontchooserdialog.run()
fontchooserdialog.destroy()
```

Note: If you application only uses a dialog window, the `Gtk.main()` call is not required. This is invoked automatically when calling `fontchooserdialog.run()`.

The title of the dialog which is displayed can be set with:

```
fontchooserdialog.set_title(title)
```

83.3 Example

Below is an example of a FontChooserDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

```

(continues on next page)

(continued from previous page)

```
fontchooserdialog = Gtk.FontChooserDialog()
fontchooserdialog.set_title("FontChooserDialog")

response = fontchooserdialog.run()

if response == Gtk.ResponseType.OK:
    print("Font selected: %s" % fontchooserdialog.get_font())

fontchooserdialog.destroy()
```

Download: `FontChooserDialog`

CHAPTER

EIGHTYFOUR

FONTBUTTON

The FontButton provides access to a *FontChooserDialog*. The button is used when space is limited to allow selecting of a font.

84.1 Constructor

The FontButton can be constructed using the following:

```
fontbutton = Gtk.FontButton(title, font)
```

When the *title* parameter has been specified, the text string is displayed on the dialog which appears when the FontButton has been clicked. Also, a *font* value can be declared to allow a font, style and size option at construction time.

84.2 Methods

The title of the dialog which is displayed can be set with:

```
fontbutton.set_title(title)
```

84.3 Signals

The commonly used signals of an FontButton are:

```
"font-set" (fontbutton)
```

A "font-set" signal emits from the FontButton when a font has been selected and the OK button has been pressed on the dialog.

If only certain elements of the font selected are required use the following methods:

```
font_face = fontbutton.get_font_face()
font_size = fontbutton.get_font_size()
font_desc = fontbutton.get_font_desc()
```

84.4 Example

Below is an example of a FontButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def font_changed(fontbutton):
    print("Font selected: %s" % fontbutton.get_font_name())

window = Gtk.Window()
window.set_default_size(150, -1)
window.connect("destroy", Gtk.main_quit)

fontbutton = Gtk.FontButton(title="FontButton")
fontbutton.connect("font-set", font_changed)
window.add(fontbutton)

window.show_all()

Gtk.main()
```

Download: `FontButton`

CHAPTER

EIGHTYFIVE

COLORCHOOSER

The ColorChooser interface is implemented by *ColorChooserDialog*, *ColorChooserWidget* and *ColorButton* to allow a user to select a colour.

85.1 Methods

By default, only colours can be selected within the ColorChooser. To enable setting of transparency call:

```
colorchooser.set_use_alpha(use_alpha)
```

When *use_alpha* is set to `True`, a slider appears within the dialog to control the amount of transparency.

To retrieve the colour from the ColorChooser use:

```
colorchooser.get_color()
colorchooser.get_rgba()
```

The `.get_color()` method returns a GdkColor object with associated values for red, green and blue. Alternatively, if your ColorChooser allows the selection of transparency values then `.get_rgba()` can be used. This also returns values for red, green and blue, and the transparency value, and is known as an GdkRGBA object. All the values returned are between `0.0` and `1.0`.

Colours can also be set specifically on the ColorChooser with:

```
colorchooser.set_color(color)
colorchooser.set_rgba(rgba)
```

The *color* and *rgba* parameters should be set to the appropriate GdkColor or GdkRGBA objects which specify the values to be used.

85.2 Signals

The common signals of the ColorChooser are:

```
"color-activated" (chooser, color)
```

The *color* value is returned when the user activates the colour in the chooser, emitting the `"color-activated"` signal.

85.3 Example

To view an example of the ColorChooser, see the code for the objects *ColorChooserWidget*, *ColorChooserDialog*, and *ColorButton*.

CHAPTER

EIGHTYSIX

COLORCHOOSERWIDGET

A ColorChooserWidget provides a dialog window from which a user can choose a colour from a palette offered.

86.1 Constructor

The ColorChooserWidget is constructed using:

```
colorchooserwidget = Gtk.ColorChooserWidget()
```

86.2 Examples

Below is an example of a ColorChooserWidget:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ColorChooserWidget(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title('ColorChooserWidget')
        self.set_border_width(5)
        self.connect('destroy', Gtk.main_quit)

        colorchooserwidget = Gtk.ColorChooserWidget()
        colorchooserwidget.connect('color-activated', self.on_color_activated)
        self.add(colorchooserwidget)

    def on_color_activated(self, colorchooserwidget, color):
        red = (color.red * 255)
        green = (color.green * 255)
        blue = (color.blue * 255)

        print('Hex: #%02x%02x%02x' % (red, green, blue))

window = ColorChooserWidget()
window.show_all()

Gtk.main()
```

Download: `ColorChooserWidget`

CHAPTER

EIGHTYSEVEN

COLORCHOOSERDIALOG

A ColorChooserDialog provides a dialog window from which a user can choose a colour from a palette offered.

87.1 Constructor

The ColorChooserDialog is constructed using:

```
colorchooserdialog = Gtk.ColorChooserDialog()
```

87.2 Methods

A title should be set on the ColorChooserDialog indicating the function of the dialog:

```
colorchooserdialog.set_title(title)
```

87.3 Examples

Below is an example of a ColorChooserDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def color_activated():
    color = colorchooserdialog.get_rgba()

    red = (color.red * 255)
    green = (color.green * 255)
    blue = (color.blue * 255)

    print('Hex: #%02x%02x%02x' % (red, green, blue))

colorchooserdialog = Gtk.ColorChooserDialog()

if colorchooserdialog.run() == Gtk.ResponseType.OK:
    color_activated()

colorchooserdialog.destroy()
```

Download: `ColorChooserDialog`

CHAPTER

EIGHTYEIGHT

COLORBUTTON

A ColorButton is used to select colour and transparency via a dialog which appears when the user clicks the button. It is commonly used in applications where space is limited.

88.1 Constructor

The ColorButton can be constructed using the following:

```
colorbutton = Gtk.ColorButton()
```

88.2 Methods

The title of the dialog which appears when clicking on the ColorButton can be set with:

```
colorbutton.set_title(title)
```

88.3 Examples

Below is an example of a ColorButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ColorButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title('ColorButton')
        self.set_default_size(200, -1)
        self.connect('destroy', Gtk.main_quit)

        colorbutton = Gtk.ColorButton()
        colorbutton.connect('color-set', self.on_color_set)
        self.add(colorbutton)

    def on_color_set(self, colorbutton):
        color = colorbutton.get_rgba()
```

(continues on next page)

(continued from previous page)

```
        red = (color.red * 255)
        green = (color.green * 255)
        blue = (color.blue * 255)

        print('Hex: #%02x%02x%02x' % (red, green, blue))

window = ColorButton()
window.show_all()

Gtk.main()
```

Download: `ColorButton`

CHAPTER

EIGHTYNINE

APPCHOOSER

AppChooser is an interface for choosing an application from a list, and is used by the widgets *AppChooserWidget*, *AppChooserDialog*, and *AppChooserButton*.

89.1 Methods

To retrieve the application information from the AppChooser use:

```
appchooser.get_app_info()
```

The method allows for the retrieval of a range of information such as name, description, and application location.

Retrieval of the content type associated with the AppChooser can be fetched by calling:

```
appchooser.get_content_type()
```

The AppChooser can be refreshed manually with the call:

```
appchooser.refresh()
```

Within the dialog, a string of text describes what action the AppChooserDialog will perform. By default this is "Select an application for (filetype) files". This can be changed using:

```
appchooserdialog.set_heading(heading)
```

89.2 Example

To view an example of the AppChooser, see the code for the objects *AppChooserWidget*, *AppChooserDialog*, and *AppChooserButton*.

CHAPTER

NINETY

APPCHOOSERWIDGET

The AppChooserWidget object allows for an application chooser to be added to a window or dialog container, allowing custom interfaces to be built.

In most cases, an *AppChooserDialog* or :doc‘appchooserbutton‘ would be used.

The Widget-variant of the AppChooser family allows for inserting into applications and custom dialogs. If space is short, use *AppChooserButton* or if a dialog is to be shown to the user use *AppChooserDialog*.

90.1 Constructor

The AppChooserWidget can be constructed using the following:

```
appchooserwidget = Gtk.AppChooserWidget(content_type)
```

A *content_type* value can be specified at construction to limit displayed applications only to those which can open the specified file.

90.2 Methods

The string of text which is displayed when there are no available applications can be set by:

```
appchooserwidget.set_default_text(default_text)
```

To set the AppChooserWidget to simply show all applications use:

```
appchooserwidget.set_show_all(show_all)
```

When *show_all* is specified as `True`, all applications are shown in a flat-layout.

To configure whether default, recommended, fallback or other applications are shown which match the mimetype call:

```
appchooserwidget.set_show_default(show_default)
appchooserwidget.set_show_recommended(show_recommended)
appchooserwidget.set_show_fallback(show_fallback)
appchooserwidget.set_show_other(show_other)
```

If *show_default*, *show_recommended*, *show_fallback* or *show_other* are set to `False`, the widget will not display those categories of application. By default, all four are set to `True`.

..note :

The AppChooserWidget utilises the *AppChooser* backend for common methods and functions.

90.3 Signals

The commonly used signals of a AppChooserWidget are:

```
"application-activated" (application)
"application-selected" (application)
```

The "application-activated" signal is emitted from the widget when the user presses the OK button, or via the keyboard when Return/Enter is pressed to select an application. Alternatively, a "application-selected" signal emits when an item within the widget is picked from the list.

90.4 Example

Below is an example of a AppChooserWidget:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class AppChooserWidget(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        appchooserwidget = Gtk.AppChooserWidget(content_type="video/webm")
        appchooserwidget.connect("application-activated", self.on_application_
↪activated)
        self.add(appchooserwidget)

    def on_application_activated(self, appchooserwidget, desktopappinfo):
        app_info = appchooserwidget.get_app_info()
        name = app_info.get_display_name()
        description = app_info.get_description()

        print("Name:\t\t%s" % (name))
        print("Description:\t%s" % (description))

window = AppChooserWidget()
window.show_all()

Gtk.main()
```

Download: AppChooserWidget

CHAPTER

NINETYONE

APPCHOOSERDIALOG

The AppChooserDialog provides a dialog container which allows selecting or opening of applications. The dialog lists applications which are associated with the specified file type.

91.1 Constructor

The AppChooserDialog can be constructed using the following:

```
appchooserdialog = Gtk.AppChooserDialog(flags, content_type)
```

The *flags* parameter should be set to `Gtk.DialogFlags.MODAL` or `Gtk.DialogFlags.DESTROY_WITH_PARENT`. The *content_type* value should be set to the mimetype of the file which is to be opened.

A *title* should be placed on the AppChooserDialog indicating the function:

```
appchooserdialog.set_title(title)
```

To ensure correct positioning of the dialog on the parent window, use:

```
appchooserdialog.set_transient_for(parent)
```

..note :

The AppChooserDialog utilises the *AppChooser* backend for common methods and functions.

91.2 Methods

When the AppChooserDialog has been created use:

```
appchooserdialog.run()
appchooserdialog.destroy()
```

91.3 Signals

The commonly used signals of a AppChooserDialog are:

```
"response" (dialog, response_id)
"close" (dialog)
```

The "close" event occurs when the user presses the Escape button on the keyboard, or the Gtk.ResponseType.CLOSE response is met. Alternatively, "response" can be emitted when anything happens within the AppChooserDialog. Both events emit the AppChooserDialog object with the function, however the "response" signal also emits a response_id value of the event that occurred within the AppChooserDialog.

91.4 Example

Below is an example of a AppChooserDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class AppChooserDialog(Gtk.AppChooserDialog):
    def __init__(self):
        Gtk.AppChooserDialog.__init__(self, content_type="image/png")
        self.set_title("AppChooserDialog")
        self.connect("response", self.on_response)

    def on_response(self, dialog, response):
        if response == Gtk.ResponseType.OK:
            app_info = appchooserdialog.get_app_info()
            name = app_info.get_display_name()
            description = app_info.get_description()

            print("Name:\t\t%s" % (name))
            print("Description:\t%s" % (description))

appchooserdialog = AppChooserDialog()
appchooserdialog.run()
```

Download: AppChooserDialog

CHAPTER

NINETYTWO

APPCHOOSERBUTTON

An AppChooserButton allows the selection of applications via a dropdown menu. The use of an AppChooserButton is useful in applications which have limited space available.

92.1 Constructor

The AppChooserButton can be constructed using the following:

```
appchooserbutton = Gtk.AppChooserButton(content_type)
```

Setting the *content_type* to a MIME type allows the AppChooserButton to limit the applications shown to those which are able to open the content type specified.

92.2 Methods

By default, the AppChooserButton is displayed as a basic dropdown menu. To enable the advanced dropdown and associated dialog functionality, from which items are chosen use:

```
appchooserbutton.set_show_dialog_item(show_dialog_item)
```

If *show_dialog_item* is set to `True`, applications matching the MIME type are displayed in a dialog.

..note :

The AppChooserButton utilises the *AppChooser* backend for common methods and functions.

92.3 Signals

The commonly used signals of a AppChooserButton are:

```
"changed" (appchooserbutton)
```

The `"changed"` signal emits from the AppChooserButton when the user changes the application which is to open the item.

92.4 Example

Below is an example of a AppChooserButton:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class AppChooserButton(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, -1)
        self.connect("destroy", Gtk.main_quit)

        appchooserbutton = Gtk.AppChooserButton(content_type="audio/flac")
        appchooserbutton.set_show_dialog_item(True)
        appchooserbutton.connect("changed", self.on_item_changed)
        self.add(appchooserbutton)

    def on_item_changed(self, appchooserbutton):
        app_info = appchooserbutton.get_app_info()
        name = app_info.get_display_name()
        description = app_info.get_description()

        print("Name:\t\t%s" % (name))
        print("Description:\t%s" % (description))

window = AppChooserButton()
window.show_all()

Gtk.main()
```

Download: `AppChooserButton`

CHAPTER

NINETYTHREE

RECENTMANAGER

The RecentManager object is the backend of the RecentChooser family, and allows for management of the files that are displayed there.

93.1 Constructor

The RecentManager can be constructed using the following:

```
recentmanager = Gtk.RecentManager()
```

93.2 Methods

Items can be added to the RecentManager by specifying the URI and calling the method:

```
recentmanager.add_item(uri)
```

They can also be removed based on the URI as well by using:

```
recentmanager.remove_item(uri)
```

Recent items can be retrieved from the RecentManager by calling:

```
items = recentmanager.get_items()
```

This method returns a list of all the URIs within the RecentManager.

To check whether the RecentManager contains a specific URI use:

```
item = recentmanager.has_item()
```

If the specified URI was found, `True` is returned.

CHAPTER

NINETYFOUR

RECENTCHOOSER

The RecentChooser is an interface for the objects *RecentChooserWidget*, *RecentChooserDialog*, and *RecentChooserMenu*. It is used to display recently opened files for quick access by the user. It is commonly used by the *FileChooser* family, however can be used in other applications.

94.1 Methods

The selected items can be fetched from the RecentChooser using:

```
item = recentchooser.get_current_item()
uri = recentchooser.get_current_uri()
```

Both the `.get_current_item()` and `.get_current_uri()` methods return a single item.

If multiple item selection is enabled, the method to use to return all selected items is:

```
items = recentchooser.get_items()
```

If a file can not be found, it may be useful to customise whether it can be viewed in the recent list:

```
recentchooser.set_show_not_found(show_not_found)
```

To configure whether only local files are displayed use:

```
recentchooser.set_local_only(local_only)
```

If there is a requirement to allow the user to select multiple items from the chooser use:

```
recentchooser.set_select_multiple(multiple)
```

When *multiple* is set to `True`, the user can hold down the `Control` key and click with the mouse.

To limit the number of items which are displayed use the method:

```
recentchooser.set_limit(limit)
```

The *limit* value should be an integer value, however it can also be set to `-1` to display all files.

The sorting type of the list can be configured by specifying:

```
recentchooser.set_sort_type(sort_type)
```

The *sort_type* should be set to one of `Gtk.RecentSortType.NONE`, `Gtk.RecentSortType.MRU` which shows most recently used at the top, and `Gtk.RecentSortType.LRU` which sorts by least recently used.

By default, the RecentChooserWidget shows icons relating to the file type. These can be disabled with:

```
recentchooser.set_show_icons(show_icons)
```

When *show_icons* is set to `False`, only the filenames will be shown.

In many cases, it may be useful to limit the files listed in the RecentChooser to those only openable by the application. This can be set using either method:

```
recentchooser.add_filter(filter)
recentchooser.set_filter(filter)
```

Filters can be removed with:

```
recentchooser.remove_filter(filter)
```

In the case of `.add_filter()`, `.set_filter()`, and `.remove_filter()`, the object should be a *RecentFilter*.

A list of RecentFilter objects attached to the RecentChooser can be found via:

```
filters = recentchooser.list_filters()
```

94.2 Signals

The common functions of the RecentChooser are:

```
"item-activated" (recentchooser)
"selection-changed (recentchooser)
```

The `"item-activated"` signal is emitted when the user either double-clicks on an item, or presses `Enter` while an item is selected. Alternatively, a `"selection-changed"` signal emits when the selection changes via the mouse of keyboard.

CHAPTER

NINETYFIVE

RECENTCHOOSERWIDGET

A RecentChooserWidget allows the selection of documents and files which have previously been opened. The widget-variant can be placed inside a Window or Dialog container.

95.1 Constructor

The RecentChooserWidget can be constructed using the following:

```
recentchooserwidget = Gtk.RecentChooserWidget(manager)
```

Setting the *manager* parameter to a *RecentManager* object allows for increase control over the content of the RecentChooserWidget.

Note: A RecentManager is not required to be added if only basic functions of the RecentChooserWidget are required. The RecentManager simply provides more options on working with files.

95.2 Methods

95.3 Example

Below is an example of a RecentChooserWidget:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class RecentChooserWidget(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title('RecentChooserWidget')
        self.set_default_size(300, 250)
        self.set_border_width(5)
        self.connect('destroy', Gtk.main_quit)

        recentchooserwidget = Gtk.RecentChooserWidget()
        recentchooserwidget.connect('item-activated', self.on_item_activated)
        self.add(recentchooserwidget)
```

(continues on next page)

(continued from previous page)

```
    def on_item_activated(self, recentchooserwidget):
        item = recentchooserwidget.get_current_item()

        if item:
            print('Item selected:')
            print('Name:\t %s' % (item.get_display_name()))
            print('URI:\t %s' % (item.get_uri()))

window = RecentChooserWidget()
window.show_all()

Gtk.main()
```

Download: `RecentChooserWidget`

CHAPTER

NINETYSIX

RECENTCHOOSERDIALOG

A RecentChooserDialog provides selection of recently opened documents from a dialog.

96.1 Constructor

The RecentChooserDialog can be constructed using the following:

```
recentchooserdialog = Gtk.RecentChooserDialog(manager, (buttons))
```

Setting the *manager* parameter to a *RecentManager* object allows for increase control over the content of the RecentChooserDialog. Finally, the *buttons* parameter should be a tuple of buttons which are to be displayed on the dialog.

Note: A RecentManager is not required to be added if only basic functions of the RecentChooserWidget are required. The RecentManager simply provides more options on working with files.

96.2 Methods

Once the RecentChooserDialog has been created, use the following to run and then destroy the widget:

```
recentchooserdialog.run()
recentchooserdialog.destroy()
```

GTK+ will loop in the `.run()` method until it receives a response, upon which any code that needs to be run is executed (for example, responding to the users request). After completion, the `.destroy()` method will remove the RecentChooserDialog.

A title can be set on the RecentChooserDialog with:

```
recentchooserdialog.set_title(title)
```

To ensure the dialog is positioned correctly over the parent window, it is recommended to set parent with:

```
recentchooserdialog.set_parent(parent)
```

96.3 Signals

The commonly used signals of a RecentChooserDialog are:

```
"response" (dialog, response_id)
"close" (dialog)
```

The "close" event occurs when the user presses the Escape button on the keyboard, or the Gtk.ResponseType.CLOSE response is met. Alternatively, "response" can be emitted when anything happens within the RecentChooserDialog. Both events emit the RecentChooserDialog object with the function, however the "response" signal also emits a response_id value of the event that occurred within the RecentChooserDialog.

96.4 Example

Below is an example of a RecentChooserDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class RecentChooserDialog(Gtk.RecentChooserDialog):
    def __init__(self):
        Gtk.RecentChooserDialog.__init__(self)
        self.set_title('RecentChooserDialog')
        self.set_default_size(250, -1)
        self.add_button('Cancel', Gtk.ResponseType.CANCEL)
        self.add_button('OK', Gtk.ResponseType.OK)
        self.set_default_response(Gtk.ResponseType.OK)
        self.connect('response', self.on_response)

    def on_response(self, recentchooserdialog, response):
        if response == Gtk.ResponseType.OK:
            item = recentchooserdialog.get_current_item()

            if item:
                print('Item selected:')
                print('Name:\t %s' % (item.get_display_name()))
                print('URI:\t %s' % (item.get_uri()))

dialog = RecentChooserDialog()
dialog.run()
dialog.destroy()
```

Download: RecentChooserDialog

CHAPTER

NINETYSEVEN

RECENTCHOOSERMENU

The RecentChooserMenu provides a list of recently opened files which are displayed via a menu.

97.1 Constructor

The RecentChooserMenu can be constructed using the following:

```
recentchoosermenu = Gtk.RecentChooserMenu()
```

97.2 Methods

To configure whether numbers are displayed on the menu, use:

```
recentchoosermenu.set_show_numbers(show_numbers)
```

When the *show_numbers* attribute is set to `False`, numbers will not be displayed next to the recently opened files. The numbers also serve as a unique accelerator key to open the files via the keyboard.

97.3 Example

Below is an example of a RecentChooserMenu:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class RecentChooserMenu(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title('RecentChooserMenu')
        self.connect('destroy', Gtk.main_quit)

        menubar = Gtk.MenuBar()
        self.add(menubar)

        menuitem = Gtk.MenuItem('Recent Items')
        menubar.append(menuitem)

```

(continues on next page)

(continued from previous page)

```
        recentchoosermenu = Gtk.RecentChooserMenu()
        recentchoosermenu.connect('item-activated', self.on_item_activated)
        menuitem.set_submenu(recentchoosermenu)

    def on_item_activated(self, recentchoosermenu):
        item = recentchoosermenu.get_current_item()

        if item:
            print("Item selected:")
            print("Name:\t %s" % (item.get_display_name()))
            print("URI:\t %s" % (item.get_uri()))

window = RecentChooserMenu()
window.show_all()

Gtk.main()
```

Download: `RecentChooserMenu`

CHAPTER

NINETYEIGHT

RECENTFILTER

A RecentFilter works in conjunction with the RecentChooser family of widgets. It is used to provide configurability over what is displayed within the RecentChooser's.

98.1 Constructor

The RecentFilter can be constructed using the following:

```
recentfilter = Gtk.RecentFilter()
```

Note: A RecentFilter must be created for each filter type needed.

98.2 Methods

To set the name which identifies the RecentFilter use:

```
recentfilter.set_name(name)
```

The name of the RecentFilter can also be retrieved with:

```
name = recentfilter.get_name()
```

Filtering by the file extension, or a particular portion of the filename can be done using:

```
recentfilter.add_pattern(pattern)
recentfilter.add_application(application)
```

The *pattern* should be set to a string of text, which matches that of the required items. To limit by extension, and example string of ".odt" is used, with the asterisk identifying a wildcard. The use of `.add_application()` allows specifying a name for an application and filtering based on that name.

Alternatively, to limit by mime type use:

```
recentfilter.add_mime_type(mime_type)
```

Limiting the RecentChooser-list by a set number of days can be set with:

```
recentfilter.add_age(days)
```

The *days* value should be set to an integer value indicating the number of days or fewer with which items should be displayed.

98.3 Example

Below is an example of a RecentFilter:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class RecentFilter(Gtk.RecentChooserDialog):
    def __init__(self):
        Gtk.RecentChooserDialog.__init__(self)
        self.set_title('RecentFilter')
        self.set_default_size(300, 200)

        recentfilter = Gtk.RecentFilter()
        recentfilter.set_name('All Items')
        recentfilter.add_pattern('*')
        self.add_filter(recentfilter)

        recentfilter = Gtk.RecentFilter()
        recentfilter.set_name('Within last 3 days')
        recentfilter.add_age(3)
        self.add_filter(recentfilter)

        recentfilter = Gtk.RecentFilter()
        recentfilter.set_name('Image Files')
        recentfilter.add_pattern('*.png')
        recentfilter.add_pattern('*.jpg')
        recentfilter.add_pattern('*.svg')
        self.add_filter(recentfilter)

dialog = RecentFilter()
dialog.run()
dialog.destroy()
```

Download: `RecentFilter`

CHAPTER

NINETYNINE

TOOLTIP

A Tooltip is a piece of text which is displayed when hovering over a widget to describe what the function of the widget is.

There are two types of tooltip; basic and advanced. Basic provides only textual information and in most cases will be used. Advanced allows displaying of styled or formatted text and icons.

99.1 Constructor

To apply a basic Tooltip to any widget, use the method:

```
widget.set_tooltip_text(text)
```

Alternatively, an advanced Tooltip can be created with:

```
tooltip = Gtk.Tooltip()
```

99.2 Methods

Note: The following methods only apply to the advanced Tooltip.

To set the text on the Tooltip use:

```
tooltip.set_text(text)
```

In some cases it may be useful to format the text with markup by calling:

```
tooltip.set_markup(markup)
```

The *markup* parameter should be set to a string, for example "Text in a Tooltip" would be displayed in bold.

Alternatively icons can also be used with:

```
tooltip.set_icon(pixbuf)
```

Specifying a *pixbuf* allows any image in the Pixbuf format to be rendered in the Tooltip.

If widgets are required to be packed into the Tooltip, then use:

```
tooltip.set_custom(custom_widget)
```

This allows packing of child widgets alongside the default Image and Label which are created at construction time.

99.3 Signals

The commonly used signals of a Tooltip are:

```
"query-tooltip" (widget, x, y, keyboard_mode, tooltip)
```

The `"query-tooltip"` parameter links to a function from which the Tooltip is called. There are several parameters passed to the function. The *widget* value identifies the widget upon which the Tooltip was called. An *x* and *y* are passed to identify the location of the cursor. The *keyboard_mode* value returns `True` or `False` depending on whether the Tooltip was called by the keyboard. Finally, the *tooltip* value indicates the Tooltip to be passed.

99.4 Examples

Below is an example of a basic Tooltip:

```
#!/usr/bin/env python3

from gi.repository import Gtk

window = Gtk.Window()
window.set_border_width(5)
window.connect("destroy", Gtk.main_quit)

label = Gtk.Label(label="Hover over this Label")
label.set_tooltip_text("This is an example of the basic Tooltip")
window.add(label)

window.show_all()

Gtk.main()
```

Download: `Tooltip Basic`

Alternatively, the advanced Tooltip is displayed below:

```
#!/usr/bin/env python3

from gi.repository import Gtk

def tooltip_query(widget, x, y, keyboard_mode, tooltip):
    tooltip.set_text("This is an example of the advanced Tooltip")

    return True

window = Gtk.Window()
window.set_border_width(5)
window.connect("destroy", Gtk.main_quit)

tooltip = Gtk.Tooltip()
```

(continues on next page)

(continued from previous page)

```
label = Gtk.Label(label="Hover over this Label")
label.set_has_tooltip(True)
label.connect("query-tooltip", tooltip_query)
window.add(label)

window.show_all()

Gtk.main()
```

Download: `Tooltip Advanced`

CHAPTER

TEXTVIEW

A TextView widget is able to display large amounts of text. It can be used to provide both editing capabilities and viewing-only functionality.

100.1 Constructor

The TextView can be constructed using the following:

```
textview = Gtk.TextView(textbuffer)
```

The *textbuffer* argument should be set to a *TextBuffer* object.

100.2 Methods

A TextBuffer can be set on the TextView with:

```
textview.set_buffer(textbuffer)
```

When constructing a TextView, a TextBuffer is automatically created however not directly usable. To name the TextBuffer and gain access call:

```
textbuffer = textview.get_buffer()
```

Setting the justification type of the text within the TextView can be configured with the method:

```
textview.set_justification(justification)
```

The *justification* value can be set to one of the following constants; `Gtk.Justification.LEFT`, `Gtk.Justification.RIGHT`, `Gtk.Justification.CENTER`, or `Gtk.Justification.FILL`.

The TextView supports a variety of wrap modes which can be configured with:

```
textview.set_wrap_mode(wrap_mode)
```

The *wrap_mode* value can be set to one of the following constants; `Gtk.WrapMode.NONE`, `Gtk.WrapMode.CHAR`, `Gtk.WrapMode.WORD` or `Gtk.WrapMode.WORDCHAR`.

Whether the TextView can be edited or not can be configured using:

```
textview.set_editable(editable)
```

When *editable* is set to `False`, the user will not be able to add or delete text within the TextView.

Another useful function is to be able to remove the cursor from view:

```
textview.set_cursor_visible(visible)
```

When using `.set_editable()`, it is recommended to also use `.set_cursor_visible()` and set it to `False` for usability reasons.

To enable overwrite mode when adding new characters to the TextView, enable with:

```
textview.set_overwrite(overwrite)
```

If *overwrite* is set to `True`, existing characters will be overwritten with new characters when typed.

Margins can be set within the TextView with:

```
textview.set_left_margin(margin)
textview.set_right_margin(margin)
textview.set_top_margin(margin)
textview.set_bottom_margin(margin)
```

The *margin* value must be set to an integer value which determines the number of pixels of space within the margin.

Indents within the TextView are defined with the methods:

```
textview.set_indent(indent)
```

The *indent* value should be an integer which indicates how the indent appears for new paragraphs of text.

Spaicing between lines can be modified by specifying the number of pixels:

```
textview.set_pixels_above_lines(pixels)
textview.set_pixels_below_lines(pixels)
```

To request that the TextView use monospaced font in the view, use the method:

```
textview.set_monospace(monospace)
```

100.3 Signals

The commonly used signals of a TextView are:

```
"move-cursor" (textview, step, count, extend_selection)
"backspace" (textview)
"select-all" (textview)
"unselect-all" (textview)
"toggle-overwrite" (textview)
"toggle-cursor-visible" (textview)
```

The TextView emits the `"move-cursor"` signal when the text cursor is moved within the text field. This includes moving the cursor via the keyboard arrows, clicking with the mouse, and pressing `Home` and `End`. Using the event passes the textview on which the move occurred, the *step* value which indicates the type of move, the *count* which specifies the number of units moved and *extend_selection* which when `True` is specified shows that the selection was extended. A `"backspace"` event is caused to emit when the `Backspace` key is used. The `"select-all"` and `"unselect-all"` events happen when the user chooses to select or unselect all of the text. When the user indicates they want to switch between insert and overwrite mode, or vice-versa, the `"overwrite-mode"` event emits. Finally, the `"toggle-cursor-visible"` signal emits from the TextView whenever the cursor is shown or hidden.

100.4 Example

Below is an example of a TextView:

```
#!/usr/bin/env python3

from gi.repository import Gtk, Pango

def set_wrap_mode(radiobutton, wrap_mode):
    textview.set_wrap_mode(wrap_mode)

def set_style_text(checkbutton):
    start, end = textbuffer.get_bounds()

    if checkbuttonBold.get_active():
        textbuffer.apply_tag(texttagBold, start, end)
    else:
        textbuffer.remove_tag(texttagBold, start, end)

    if checkbuttonItalic.get_active():
        textbuffer.apply_tag(texttagItalic, start, end)
    else:
        textbuffer.remove_tag(texttagItalic, start, end)

    if checkbuttonUnderline.get_active():
        textbuffer.apply_tag(texttagUnderline, start, end)
    else:
        textbuffer.remove_tag(texttagUnderline, start, end)

window = Gtk.Window()
window.set_default_size(250, 300)
window.connect("destroy", Gtk.main_quit)

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

scrolledwindow = Gtk.ScrolledWindow()
scrolledwindow.set_policy(Gtk.PolicyType.AUTOMATIC, Gtk.PolicyType.AUTOMATIC)
grid.attach(scrolledwindow, 0, 0, 2, 1)

textbuffer = Gtk.TextBuffer()
texttagBold = textbuffer.create_tag("Bold", weight=Pango.Weight.BOLD)
texttagItalic = textbuffer.create_tag("Italic", style=Pango.Style.ITALIC)
texttagUnderline = textbuffer.create_tag("Underline", underline=Pango.Underline.
↪SINGLE)
textbuffer.set_text("GTK+, or the GIMP Toolkit, is a multi-platform toolkit for␣
↪creating graphical user interfaces. Offering a complete set of widgets, GTK+ is␣
↪suitable for projects ranging from small one-off tools to complete application␣
↪suites.")

textview = Gtk.TextView(buffer=textbuffer)
textview.set_vexpand(True)
textview.set_hexpand(True)
scrolledwindow.add(textview)

radiobuttonWrapNone = Gtk.RadioButton(group=None, label="None")
radiobuttonWrapNone.connect("toggled", set_wrap_mode, Gtk.WrapMode.NONE)
```

(continues on next page)

(continued from previous page)

```
grid.attach(radiobuttonWrapNone, 0, 1, 1, 1)

radiobuttonWrapChar = Gtk.RadioButton(group=radiobuttonWrapNone, label="Character")
radiobuttonWrapChar.connect("toggled", set_wrap_mode, Gtk.WrapMode.CHAR)
grid.attach(radiobuttonWrapChar, 0, 2, 1, 1)

radiobuttonWrapWord = Gtk.RadioButton(group=radiobuttonWrapNone, label="Word")
radiobuttonWrapWord.connect("toggled", set_wrap_mode, Gtk.WrapMode.WORD)
grid.attach(radiobuttonWrapWord, 0, 3, 1, 1)

radiobuttonWrapWordChar = Gtk.RadioButton(group=radiobuttonWrapNone, label="Word &␣
↪Character")
radiobuttonWrapWordChar.connect("toggled", set_wrap_mode, Gtk.WrapMode.WORD_CHAR)
grid.attach(radiobuttonWrapWordChar, 0, 4, 1, 1)

checkbuttonBold = Gtk.CheckButton(label="Bold")
checkbuttonBold.connect("toggled", set_style_text)
grid.attach(checkbuttonBold, 1, 1, 1, 1)

checkbuttonItalic = Gtk.CheckButton(label="Italic")
checkbuttonItalic.connect("toggled", set_style_text)
grid.attach(checkbuttonItalic, 1, 2, 1, 1)

checkbuttonUnderline = Gtk.CheckButton(label="Underline")
checkbuttonUnderline.connect("toggled", set_style_text)
grid.attach(checkbuttonUnderline, 1, 3, 1, 1)

window.show_all()

Gtk.main()
```

Download: `TextView`

CHAPTER

ONE

TEXTBUFFER

A TextBuffer is a backend object which is used in conjuction with a *TextView*. It allows for text to be stored and shared across multiple TextView widgets.

101.1 Constructor

The TextTag can be constructed using:

```
textbuffer = Gtk.TextBuffer()
```

Note: A TextBuffer object is automatically created when using a TextView, and does not need to be constructed separately in most cases.

101.2 Methods

Applying text to the TextBuffer can be done directly using:

```
textbuffer.set_text(text, length)
```

The *text* argument should be the string of text which is to be added. Setting the *length* argument allows limiting the string to a set number of characters. This can be set to `-1` if no limit is required.

The text stored in the buffer may be retrieved with:

```
text = textbuffer.get_text(start, end, include_hidden_chars)
```

The *start* and *end* arguments should be set to TextIter objects which specify the range of text to retrieve. The *include_hidden_chars* parameter when set to `True` allows hidden characters to also be retrieved.

Retrieving the entire boundary of text can be completed with:

```
start, end = textbuffer.get_bounds()
```

The *start* and *end* values returned are TextIter objects which identify the start and end locations of the content.

Alternatively, to retrieve a selection that a user has made issue the method:

```
start, end = textbuffer.get_selection_bounds()
```

As with the `.get_bounds()` method, this returns a *start* and *end* TextIter object which identifies the selection of the text.

The TextIter start and end objects can also be retrieved individually:

```
start = textbuffer.get_start_iter()
end = textbuffer.get_end_iter()
```

Checking whether the TextBuffer has a selection can be done via:

```
has_selection = textbuffer.get_has_selection()
```

To retrieve the number of words and characters contained within the buffer use:

```
word_count = textbuffer.get_word_count()
char_count = textbuffer.get_char_count()
```

Checking whether a TextBuffer has been modified can be done with:

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

To set whether the TextBuffer has been changed use:

```
textbuffer.set_modified(modified)
```

Setting the *modified* argument to `False` is commonly used after a document has been saved and the TextBuffer status needs to be set back to unmodified.

CHAPTER

TWO

TEXTMARK

A TextMark is an object used within a *TextBuffer* to preserve a location within text. It remains valid across changes to the buffer, including insertion or deletion of text around the TextMark.

102.1 Constructor

The TextMark can be constructed using the following:

```
textmark = Gtk.TextMark(name, left_gravity)
```

The *name* parameter should specify a unique name which identifies the TextMark. Alternatively, it can be set to `None` to provide an anonymous TextMark. The *left_gravity* attribute should be a Boolean value, and when set to `False`, forces the TextMark to the right when text is inserted.

102.2 Methods

To enable a TextMark to be visible via a horizontal line placed in the text, use:

```
textmark.set_visible(visible)
```

By default, TextMark objects are not visible, however specifying `True` will show them within the TextView.

We can also check on the visibility status of a TextMark via:

```
visible = textmark.get_visible()
```

To retrieve the name of a TextMark call:

```
name = textmark.get_name()
```

If the TextMark wasn't given a name at construction time, `None` will be returned as the name.

A check on whether a TextMark has been deleted can be made using:

```
deleted = textmark.get_deleted()
```

If `True` is returned, the TextMark has been deleted.

CHAPTER

THREE

TEXTTAG

A TextTag can be applied to text contained within a *TextBuffer*. It allows many properties to be applied to the text marked by the TextTag.

TextTag objects are applied to a *TextTagTable*.

103.1 Constructor

The TextTag can be constructed using:

```
texttag = Gtk.TextTag(name)
```

A *name* must be applied to the TextTag to identify it. Ideally, it should be descriptive of the properties that the TextTag affects.

103.2 Methods

TextTag objects use priorities to control in which order properties are applied:

```
texttag.set_priority(priority)
```

The *priority* is an integer with `0` identifying the most important, and the maximum being the number of tags within the TextTagTable minus one.

Retrieving the priorty can be completed with:

```
priority = texttag.get_priority()
```

The properties should then be applied to the TextTag via:

```
texttag.set_property("property", value)
```

The *property* value identifies the change the TextTag will make, whereas *value* is used to set how much the property will change.

103.3 Properties

The most common properties that are used with a TextTag are:

- `"font"` - a textual string identifying the font type to use

- "editable" - when the value set to True, allows the user to edit the text within the cell
- "justification" - can be set to one of Gtk.Justification.LEFT, Gtk.Justification.RIGHT, Gtk.Justification.CENTER, or Gtk.Justification.FILL
- "left-margin" - a integer value specifying the number of pixels of margin width
- "right-margin" - a integer value specifying the number of pixels of margin width
- "size" - an integer providing the text size in Pango units
- "size-points" - decimal value which sets the text size in points
- "strikethrough" - specifying this value as True places a line through the selected text
- "underline" - setting to True places a line under the specified text
- "wrap-mode" - describes the line wrapping when set to either Gtk.WrapMode.NONE, Gtk.WrapMode.CHAR, Gtk.WrapMode.WORD, or Gtk.WrapMode.WORD_CHAR

CHAPTER

FOUR

TEXTTAGTABLE

A TextTagTable is an invisible object which is used to hold *TextTag* objects.

104.1 Constructor

The TextTagTable can be constructed using:

```
texttagtable = Gtk.TextTagTable()
```

104.2 Methods

TextTag objects can be added and removed from the TextTagTable using:

```
texttagtable.add(texttag)
texttagtable.remove(texttag)
```

To find a TexTag within the table, use the method:

```
texttagtable.lookup(name)
```

Returning the number of TextTag objects contained within the table can be done with:

```
size = texttagtable.get_size()
```

104.3 Signals

The commonly used signals of a TextTagTable are:

```
"tag-added" (texttagtable, tag)
"tag-removed" (texttagtable, tag)
"tag-changed" (texttagtable, tag, size_changed)
```

The `"tag-added"` and `"tag-removed"` signals emit when a TextTag is added or removed from the table. Both return the name of the TextTag added or removed. The `"tag-changed"` signal gets emitted when tags are changed. If the change affects the number of tags contained in the table, the *size_changed* parameter returns `True`.

CHAPTER

FIVE

LISTSTORE

A ListStore is a flat-layout storage model which allows the storage of a range of data including textual strings, integer and floating values, images and Boolean values. It is used as the storage area for a range of widgets including TreeView, IconView and ComboBox.

105.1 Constructor

The ListStore can be constructed using the following:

```
liststore = Gtk.ListStore(data_types)
```

The *data_types* parameter can be set to a number of supported types. These includes those from Python; `str`, `float`, `long`, `int`, `bool`, and `object`. Alternatively, GTK provides the following; `gchar`, `guchar`, `gboolean`, `gint`, `guint`, `glong`, `gulong`, `gint64`, `guint64`, `gfloat`, `gdouble`, `gchararray`, and `GObject`.

Note: There is no preference over which data type to use, however for simplicity it is often easier to use the Python data types.

To store data in the ListStore, the type of data being entered must match the data type specified. For example, if you attempt to insert an integer into a string, you will receive an error.

105.2 Methods

Values can be inserted into the ListStore in a few ways:

```
liststore.insert([data], position)
liststore.append([data])
liststore.prepend([data])
```

The `.insert()` method requires two arguments; the *position* of where to enter the data and a list containing the data. The `.append()` and `.prepend()` methods add data to the back or front of the ListStore depending on the order the code is written.

An item can be removed by calling:

```
liststore.remove(treeiter)
```

The *treeiter* parameter should be a TreeIter object retrieved using the methods specified on the page.

The values currently held in the ListStore can be cleared using:

```
liststore.clear()
```

Values contained within the ListStore can swap position with:

```
liststore.swap(position_a, position_b)
```

The *position_a* and *position_b* arguments should be TreeIter objects relating to each of the rows which are to be swapped.

To reorder all the items in a ListStore call:

```
liststore.reorder([positions])
```

The *positions* list passes the new position of each child. The method only works on unsorted ListStore objects.

An item in the ListStore can be moved before or after another item with the methods:

```
liststore.move_before(treeiter, position)
liststore.move_after(treeiter, position)
```

The *treeiter* specifies the TreeIter object of the item to be moved, with the *position* value indicating the position of the item to be moved before or after.

A TreeIter can be checked to ensure it is valid with the method:

```
liststore.get_iter_is_valid(treeiter)
```

If the method returns `False`, the passed TreeIter is not valid in the ListStore.

105.3 Example

Below is an example of a ListStore:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ListStore(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        liststore = Gtk.ListStore(str, int)
        liststore.append(["Oranges", 5])
        liststore.append(["Apples", 3])
        liststore.append(["Bananas", 2])
        liststore.append(["Tomatoes", 4])
        liststore.append(["Cucumber", 1])

        treeview = Gtk.TreeView()
        treeview.set_model(liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Item")
```

(continues on next page)

(continued from previous page)

```
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        treeviewcolumn = Gtk.TreeViewColumn("Quantity")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 1)

window = ListStore()
window.show_all()

Gtk.main()
```

Download: `ListStore`

Note: The above example makes use of *TreeView* and *CellRendererText* widgets, which will be covered in later chapters.

CHAPTER

SIX

TREESTORE

A TreeStore is a data store object which allows data to be stored in a multi-level structure.

106.1 Constructor

The TreeStore can be constructed using the following:

```
treestore = Gtk.TreeStore(data_types)
```

The *data_types* parameter can be set to a number of supported types. These includes those from Python; `str`, `float`, `long`, `int`, `bool`, and `object`. Alternatively, GTK provides the following; `gchar`, `guchar`, `gboolean`, `gint`, `guint`, `glong`, `gulong`, `gint64`, `guint64`, `gfloat`, `gdouble`, `gchararray`, and `GObject`.

Note: There is no preference over which data type to use, however for simplicity it is often easier to use the Python data types.

To store data in the TreeStore, the type of data being entered must match the data type specified.

106.2 Methods

Insertion of a row is made using the methods:

```
treestore.insert(parent, [data], positon)
treestore.append(parent, [data])
treestore.prepend(parent, [data])
```

The `.insert()` method takes a *position* parameter which allows the row to be added to the TreeStore in a particular location. Each method takes the *parent* which can be `None` if the data has no parent, or it can specify the TreeIter of the parent which the data will be attached to.

To remove an item from the TreeStore use:

```
treestore.remove(treeiter)
```

The *treeiter* value should be a TreeIter retrieved using the methods described on that page.

Emptying the TreeStore of all values can be done with the method:

```
treestore.clear()
```

Values contained within the TreeStore can swap position with:

```
treestore.swap(position_a, position_b)
```

The *position_a* and *position_b* arguments should be TreeIter objects relating to each of the rows which are to be swapped.

To reorder all the items in a TreeStore call:

```
treestore.reorder(parent, [positions])
```

The *parent* parameter specifies the parent TreeIter of the items to be reordered. The *positions* list passes the new position of each child. The method only works on unsorted TreeStore objects.

An item in the TreeStore can be moved before or after another item with the methods:

```
treestore.move_before(treeiter, position)
treestore.move_after(treeiter, position)
```

The *treeiter* specifies the TreeIter object of the item to be moved, with the *position* value indicating the position of the item to be moved before or after.

A TreeIter can be checked to ensure it is valid with the method:

```
treestore.get_iter_is_valid(treeiter)
```

If the method returns `False`, the passed TreeIter is not valid in the TreeStore.

Obtaining the depth of a passed TreeIter can be found with:

```
treestore.iter_depth(treeiter)
```

The method will return `0` if the TreeIter is a top level item, with the depth value returned for each item down the tree.

106.3 Example

Below is an example of a TreeStore:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class TreeStore(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, -1)
        self.connect("destroy", Gtk.main_quit)

        treestore = Gtk.TreeStore(str)
        dog = treestore.append(None, ["Dog"])
        treestore.append(dog, ["Fido"])
        treestore.append(dog, ["Spot"])
        cat = treestore.append(None, ["Cat"])
        treestore.append(cat, ["Ginger"])
        rabbit = treestore.append(None, ["Rabbit"])
        treestore.append(rabbit, ["Twitch"])
        treestore.append(rabbit, ["Floppy"])
```

(continues on next page)

(continued from previous page)

```
        treeview = Gtk.TreeView()
        treeview.set_model(treestore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Pet Names")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

window = TreeStore()
window.show_all()

Gtk.main()
```

Download: `TreeStore`

Note: The above example makes use of *TreeView* and *CellRendererText* widgets, which will be covered in later chapters.

CHAPTER

SEVEN

COMBOBOX

A ComboBox provides a drop-down menu of items which can be selected by the user. It is commonly used when there are more than five items to choose from.

Another function is to provide a way to enter values via the keyboard if the option is required.

107.1 Constructor

The ComboBox can be constructed using the following:

```
combobox = Gtk.ComboBox(model)
```

A *model* should be assigned to the ComboBox to provide the list of options which are selectable.

ComboBox widgets can also be created with an Entry and/or associated model via:

```
combobox = Gtk.ComboBox.new_with_entry()
combobox = Gtk.ComboBox.new_with_model(model)
combobox = Gtk.ComboBox.new_with_model_and_entry(model)
```

107.2 Methods

To set the model on the ComboBox after constructing the widget use:

```
combobox.set_model(model)
```

The model is also retrievable:

```
combobox.get_model()
```

In order to return the currently selected item from the ComboBox call:

```
combobox.get_active()
```

The *active* value will return the number identifying the item. The first item will return `0` as the index number. If no item is selected, `-1` will be returned.

Alternatively, to set the item on the ComboBox use:

```
combobox.set_active(active)
```

A TreeIter object representing the item in the ComboBox can be retrieved via:

```
combobox.get_active_iter()
```

The selected row can also be set based on the TreeIter by:

```
combobox.set_active_iter(treeiter)
```

An id column allows items to be selected based on a unique string value. The column in the model which is to be used for the id is set using:

```
combobox.set_id_column(column)
```

The active id in use can be retrieved via:

```
combobox.get_active_id()
```

Also, the id can be set by calling:

```
combobox.set_active_id(id)
```

If the ComboBox contains a large number of items, it is recommended to display the menu spanning multiple columns. This can be set with:

```
combobox.set_wrap_width(width)
```

The *width* value should be set to the number of columns required.

The number of rows and columns an item should span can be set with the methods:

```
combobox.set_row_span_column(row_span)
combobox.set_column_span_column(column_span)
```

The *row_span* and *column_span* methods should be set to the column held in the model, which contains an integer detailing the number of rows and columns the item will span when displayed.

To check whether a ComboBox has an associated text entry:

```
combobox.get_has_entry()
```

Text displayed in the dropdown can be assigned a column from the model by:

```
combobox.set_entry_text_column(column)
```

107.3 Signals

The commonly used signals of a ComboBox are:

```
"changed" (combobox)
"popup" (combobox)
"popdown" (combobox)
```

A `"changed"` event occurs when the selected item within the ComboBox is changed. The signals `"popup"` and `"popdown"` are emitted when the menu object of the Combobox is shown or hidden by the user.

107.4 Example

Below is an example of a ComboBox:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ComboBox(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(150, -1)
        self.connect("destroy", Gtk.main_quit)

        liststore = Gtk.ListStore(str)

        for item in ["Debian", "Fedora", "Tiny Core", "Linux Mint", "Mageia"]:
            liststore.append([item])

        combobox = Gtk.ComboBox()
        combobox.set_model(liststore)
        combobox.set_active(0)
        combobox.connect("changed", self.on_combobox_changed)
        self.add(combobox)

        cellrenderertext = Gtk.CellRendererText()
        combobox.pack_start(cellrenderertext, True)
        combobox.add_attribute(cellrenderertext, "text", 0)

    def on_combobox_changed(self, combobox):
        treeiter = combobox.get_active_iter()
        model = combobox.get_model()

        print("ComboBox selected item: %s" % (model[treeiter][0]))

window = ComboBox()
window.show_all()

Gtk.main()
```

Download: `ComboBox`

CHAPTER

EIGHT

COMBOBOXTEXT

A ComboBoxText provides a simple variant of *ComboBox* which is suited to text-only values.

108.1 Constructor

The ComboBoxText can be constructed using the following:

```
comboboxtext = Gtk.ComboBoxText()
```

Alternatively, if you wish to allow the user to enter text as well as selecting from the pre-defined list, use:

```
comboboxtext = Gtk.ComboBoxText.new_with_entry()
```

108.2 Methods

Items can be added to the dropdown with the following methods:

```
comboboxtext.insert(position, id, text)
comboboxtext.append(id, text)
comboboxtext.prepend(id, text)
```

The *id* defines a unique string which identifies the item in the ComboBoxText. The *text* value provides the string to be displayed to the end user. Finally, the *position* on the `.insert()` methods sets the location the new item should be added.

If the ComboBoxText is simply used for listing text items, and id values are not required, use:

```
comboboxtext.insert_text(position, text)
comboboxtext.append_text(text)
comboboxtext.prepend_text(text)
```

All the items can be removed from the dropdown menu:

```
comboboxtext.remove_all()
```

Alternatively, single items can be deleted base on their listed position in the dropdown menu:

```
comboboxtext.remove(position)
```

The *position* parameter is the location where the item is located within the dropdown, with `0` indicating the first item.

To retrieve the text which has been selected call:

```
comboboxtext.get_active_text()
```

Alternatively, the active item id can be retrieved via:

```
comboboxtext.get_active_id()
```

TreeIter objects can also be fetched from the ComboBoxText by using the method:

```
comboboxtext.get_active_iter()
```

Setting a default selected item can be achieved with:

```
comboboxtext.set_active(position)
```

The *position* value must be an integer number representing the location of the item in the dropdown menu.

The active id can also be defined:

```
comboboxtext.set_active_id(id)
```

A row may also be selected based on a known TreeIter with:

```
comboboxtext.set_active_iter(treeiter)
```

108.3 Example

Below is an example of a ComboBoxText:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class ComboBoxText(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(150, -1)
        self.connect("destroy", Gtk.main_quit)

        comboboxtext = Gtk.ComboBoxText()
        comboboxtext.append("gnome", "GNOME")
        comboboxtext.append("kde", "KDE")
        comboboxtext.append("xfce", "XFCE")
        comboboxtext.append("lxde", "LXDE")
        comboboxtext.set_active_id("xfce")
        comboboxtext.connect("changed", self.on_comboboxtext_changed)
        self.add(comboboxtext)

    def on_comboboxtext_changed(self, comboboxtext):
        print("ComboBox selected item: %s" % (comboboxtext.get_active_text()))

window = ComboBoxText()
window.show_all()
```

(continues on next page)

(continued from previous page)

```
Gtk.main()
```

Download: `ComboBoxText`

CHAPTER

NINE

TREEVIEW

A TreeView is a table-style widget that allows for displaying information in columns and rows. It can be used to display a variety of information in different styles through the user of cell types. The TreeView is the most advanced widget to implement and has a high-learning curve.

109.1 Constructor

The TreeView can be constructed using the following:

```
treeview = Gtk.TreeView(model)
```

The *model* parameter should be set to either a *ListStore* or *TreeStore*, depending on the data which is to be displayed.

109.2 Methods

The model can also be set after construction using:

```
treeview.set_model(model)
```

Attaching *TreeViewColumn* objects to the TreeView can be done in a number of ways:

```
treeview.append_column(column)
treeview.prepend_column(column)
treeview.insert_column(column, position)
```

The *column* parameter should be set to the column name which is to be attached. The *position* parameter within the `.insert_column()` method allows the specification of a position value where the column should be added, with `0` indicating the first place.

A column can also be removed if necessary with:

```
treeview.remove_column(column)
```

To fetch the number of columns within the TreeView:

```
n_columns = treeview.get_n_columns()
```

By default, the column headers will be visible. These can be disabled with:

```
treeview.set_headers_visible(headers_visible)
```

When *headers_visible* is set to `False`, the column headers will not be shown even if there is a title set on them.

The TreeView supports basic searching of data. This can be enabled with:

```
treeview.set_enable_search(enable_search)
```

If *enable_search* is set to `True`, when the user begins typing a text entry appears with the characters to issue the search by.

When setting the `.set_enable_search()` method to `True`, the first column will be used to test the search string against. This can be modified with:

```
treeview.set_search_column(search_column)
```

The *search_column* should be set to the integer value of the column which is to be searched.

109.3 Example

For an example of the TreeView widget, see either the documentation for *ListStore* or *TreeStore*.

CHAPTER

TREEVIEWCOLUMN

A TreeViewColumn can be packed into a *TreeView*. It can display a header to describe the content of the column, from which items can be placed within to display the content.

110.1 Constructor

The TreeViewColumn can be constructed using the following:

```
treeviewcolumn = Gtk.TreeViewColumn(title)
```

The *title* property is a textual string which is attached to the top of the TreeView and describes the content.

110.2 Methods

The column title can also be specified after construction by:

```
treeviewcolumn.set_title(title)
```

Setting the *title* argument is always useful, however the text will only be displayed in the TreeView permits it.

Items can be packed into the column via two methods:

```
treeviewcolumn.pack_start(cellrenderer, expand)
treeviewcolumn.pack_end(cellrenderer, expand)
```

The *cellrenderer* property should be the name of the CellRenderer which is to be added to the column. The *expand* property can be set to `True` which expands the column to fill all available space, or `False` which shrinks it to fit the content only.

All the items from a column can be removed using:

```
treeviewcolumn.clear()
```

When multiple CellRenderer widgets are packed into a column, they have no spacing between. This can be modified with:

```
treeviewcolumn.set_spacing(spacing)
```

Configuring whether a column is visible can be set with:

```
treeviewcolumn.set_visible(visible)
```

When *visible* is `False`, the column is removed from view however still retains any settings specified.

By default, columns can not be resized. Changing this allows the user to set a custom width:

```
treeviewcolumn.set_resizable(resizable)
```

The sizing of the column can also be customised in a number of ways depending on the change in the content by using the method:

```
treeviewcolumn.set_sizing(sizing)
```

The *sizing* argument can be set to `Gtk.TreeViewColumnSizing.GROW_ONLY` sets the column to never shrink regardless of the content, `Gtk.TreeViewColumnSizing.AUTOSIZE` adjusts the column to be an optimal size and is updated everytime the model changes, and `Gtk.TreeViewColumnSizing.FIXED` which sets columns to be a fixed pixel width.

Minimum and maximum column widths can be specified by calling:

```
treeviewcolumn.set_min_width(min_width)
treeviewcolumn.set_max_width(max_width)
```

Columns within the TreeView can be moved by specifying on each:

```
treeviewcolumn.set_reorderable(reorderable)
```

CHAPTER

ONE

TREESELECTION

A TreeSelection is an object that allows for management of selections within a *TreeView*.

111.1 Methods

To set the type of selection which the TreeSelection allows, call:

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

The *mode* can be set to one of the following; `Gtk.SelectionType.NONE` which prevents a selection, `Gtk.SelectionType.SINGLE` that allows only a none or one item to be selected, `Gtk.SelectionType.BROWSE` enforces a single item to be selected, or `Gtk.SelectionType.MULTIPLE` which allows selecting of multiple items by holding the `Control` key or by click-and-drag. By default, `Gtk.SelectionType.SINGLE` is the default selection mode.

The selection mode can be gathered with the method:

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

Retrieval of the selected item when `Gtk.SelectionType.SINGLE` or `Gtk.SelectionType.BROWSE` is used can be made with:

```
selected = treeselection.get_selected(model, treeiter)
```

The *model* value should be set to the *ListStore* or *TreeStore* of the current data store. The *treeiter* value also identifies the specific row which is selected.

To select or unselect all the items in the TreeSelection call:

```
treeselection.select_all()
treeselection.unselect_all()
```

Counting the number of selected rows which the TreeSelection has can be done with the method:

```
count = treeselection.count_selected_rows()
```

The TreeView which is associated with the TreeSelection can be found via:

```
treeview = treeselection.get_tree_view()
```

111.2 Signals

The common signals of the TreeSelection are

"changed" (treeselection)

When the currently selected items within the TreeSelection are changed, the `"changed"` signal is emitted.

CHAPTER

TWO

CELLRENDERER

The CellRenderer object provides a base for the other members of the CellRenderer family. It provides common methods and properties which may be useful when displaying information in a *TreeView* or *ComboBox*.

112.1 Methods

The visibility of a particular can be toggled with:

```
cellrenderer.set_visible(visible)
```

When the *visible* argument is set to `False`, the CellRenderer will be hidden.

Sensitivity of a CellRenderer can also be changed via:

```
cellrenderer.set_sensitive(sensitive)
```

Padding can be defined for the width and height by specifying the required value in pixels:

```
cellrenderer.set_padding(xpadding, ypadding)
```

Content within the CellRenderer can be aligned by calling the method:

```
cellrenderer.set_alignment(xalign, yalign)
```

A fixed width and height can also be set by using:

```
cellrenderer.set_fixed_size(width, height)
```

The *width* and *height* can be set to `-1` to revert to automatic sizing.

CHAPTER

THREE

CELLRENDERERTEXT

A CellRendereText is used to display text-only within a *TreeView*. Besides basic text, there are a number of properties which can be used to control the look of the text including the font, style, size, foreground and background colours, and whether the text is editable.

Note: See the *CellRenderer* page for additional methods available.

113.1 Constructor

The CellRendererText can be constructed using the following:

```
cellrenderertext = Gtk.CellRendererText()
```

113.2 Methods

CellRenderer widgets only use one method which is used to set the styling functions of the cell:

```
cellrenderertext.set_property(property, value)
```

113.3 Properties

The configuration of the CellRendererText is made using the property functions:

```
cellrenderertext.set_property("item", value)
```

The property items available for use with the CellRendererText are:

- `"editable"` - when the value set to `True`, allows the user to edit the text within the cell.
- `"family"` - allows a family of fonts to be specified in order of preference, for example; cantarell, “droid sans”, monospace.
- `"font"` - a font other than the system default can be used by specifying the name and size as the value.
- `"size"` - specifies a font size in points as an integer value.
- `"strikethrough"` - specifying `True` as the value adds a strikethrough to any text within the cell.

- "text" - setting the value to the column number of the model indicates which column the text should be pulled from.
- "underline" - using True will place a line beneath the text.
- "weight" - allows configuration of letter thinness or thickness, with an integer value of 400 being standard. Higher values increase thickness, lower values decrease thickness.

113.4 Signals

The commonly used signals of a CellRendererText are:

```
"edited" (cellrenderertext, path, text)
```

An "edited" event is emitted from the CellRendererText when the user double-clicks on the cell. At this point, the cell is passed to the function along with a path identifying the location of the edited text, and finally the new text which has been entered.

113.5 Example

Below is an example of a CellRendererText:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class CellRendererText(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

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

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Distribution")
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)
        treeview.append_column(treeviewcolumn)

        cellrenderertext = Gtk.CellRendererText()
        cellrenderertext.set_property("editable", True)
        cellrenderertext.connect("edited", self.on_cell_edited)

        treeviewcolumn = Gtk.TreeViewColumn("Website")
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 1)
```

(continues on next page)

(continued from previous page)

```
        treeview.append_column(treeviewcolumn)

    def on_cell_edited(self, cellrenderertext, treepath, text):
        self.liststore[treepath][1] = text

window = CellRendererText()
window.show_all()

Gtk.main()
```

Download: `CellRendererText`

CHAPTER

FOUR

CELLRENDERERTOGGLE

When using a CellRendererToggle, it allows a widget similar to a *CheckButton* or *RadioButton* to be displayed within the TreeView.

Note: See the *CellRenderer* page for additional methods available.

114.1 Constructor

The CellRendererToggle can be constructed using the following:

```
cellrenderertoggle = Gtk.CellRendererToggle()
```

114.2 Methods

CellRenderer widgets only use one method which is used to set the styling functions of the cell:

```
cellrenderertoggle.set_property(property, value)
```

By default, the CellRendererToggle is drawn as a CheckButton. This can be changed to a RadioButton using:

```
cellrenderertoggle.set_radio(radio)
```

When *radio* is set to `True`, the RadioButton style is drawn.

To make a CellRendererToggle set as active:

```
cellrenderertoggle.set_active(active)
```

When the *active* parameter is set to `True`, the CellRendererToggle will be shown in the ticked (active) state.

To prevent a CellRendererToggle from being activated:

```
cellrenderertoggle.set_activatable(activatable)
```

114.3 Properties

The configuration of the CellRendererSpin is made using the property functions:

```
cellrendererspin.set_property("item", value)
```

The property items available for use with the CellRendererSpin are:

- `"activatable"` - customise whether the CellRendererToggle is activatable.
- `"active"` - this toggles the state of the CellRendererToggle.
- `"inconsistent"` - when set to `True`, the CellRendererToggle can be used to indicate the status of other features.
- `"radio"` - if set to `True`, the CellRendererToggle will be drawn like a RadioButton.

114.4 Signals

The commonly used signals of a CellRendererToggle are:

```
"toggled" (cellrenderertoggle, path)
```

The `"toggled"` signals emits from the CellRendererToggle when the user clicks to display or remove the tick within the widget. It provides the *path* which identifies the location of the item which has been modified.

114.5 Example

Below is an example of a CellRendererToggle:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class CellRendererToggle(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        self.liststore = Gtk.ListStore(str, bool)
        self.liststore.append(["Ethernet", True])
        self.liststore.append(["Wireless", True])
        self.liststore.append(["Bluetooth", False])
        self.liststore.append(["3g Mobile", True])

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststore)
        self.add(treeview)

        cellrendererertext = Gtk.CellRendererText()

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

        treeviewcolumn = Gtk.TreeViewColumn("Connection Type")
        treeviewcolumn.pack_start(cellrendererertext, False)
        treeviewcolumn.add_attribute(cellrendererertext, "text", 0)
        treeview.append_column(treeviewcolumn)
```

(continues on next page)

(continued from previous page)

```
        treeviewcolumn = Gtk.TreeViewColumn("Status")
        treeviewcolumn.pack_start(cellrenderertoggle, False)
        treeviewcolumn.add_attribute(cellrenderertoggle, "active", 1)
        treeview.append_column(treeviewcolumn)

    def on_cell_toggled(self, cellrenderertoggle, treepath):
        self.liststore[treepath][1] = not self.liststore[treepath][1]

window = CellRendererToggle()
window.show_all()

Gtk.main()
```

Download: `CellRendererToggle`

CHAPTER

FIVE

CELLRENDERERSPINNER

The CellRendererSpinner provides a *Spinner* widget within a TreeView, to indicate activity of a job.

Note: See the *CellRenderer* page for additional methods available.

115.1 Constructor

The CellRendererSpinner can be constructed using the following:

```
cellrendererspinner = Gtk.CellRendererSpinner()
```

115.2 Properties

The configuration of the CellRendererSpin is made using the property functions:

```
cellrendererspin.set_property("item", value)
```

The property items available for use with the CellRendererSpin are:

- `"active"` - the active property configures whether the CellRendererSpinner is displayed in the cell or not.
- `"pulse"` - the pulse value is set to indicate whether the animation is running. This would be updated twelve times in 750 milliseconds.
- `"size"` - the size property can be specified to indicate the size of the Spinner widget. The value should be set to one of the following; `Gtk.IconSize.INVALID`, `Gtk.IconSize.MENU`, `Gtk.IconSize.SMALL_TOOLBAR`, `Gtk.IconSize.LARGE_TOOLBAR`, `Gtk.IconSize.BUTTON`, `Gtk.IconSize.DND`, or `Gtk.IconSize.DIALOG`.

115.3 Example

Below is an example of a CellRendererSpinner:

```
#!/usr/bin/env python3

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

(continues on next page)

(continued from previous page)

```
class CellRendererSpinner(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        self.liststore = Gtk.ListStore(str, bool, int)
        self.liststore.append(["Copying files", True,  0])
        self.liststore.append(["Downloading access logs", False, 0])
        self.liststore.append(["Connecting to server", True, 0])

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()
        self.cellrendererspinner = Gtk.CellRendererSpinner()

        treeviewcolumn = Gtk.TreeViewColumn("Activity")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, False)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        treeviewcolumn = Gtk.TreeViewColumn("Status")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(self.cellrendererspinner, False)
        treeviewcolumn.add_attribute(self.cellrendererspinner, "active", 1)

    def on_spinner_pulse(self):
        for item in self.liststore:
            if item[1]:
                if item[2] == 12:
                    item[2] = 0
                else:
                    item[2] += 1

        self.cellrendererspinner.set_property("pulse", item[2])

        return True

window = CellRendererSpinner()
window.show_all()

GObject.timeout_add(100, window.on_spinner_pulse)

Gtk.main()
```

Download: `CellRendererSpinner`

CHAPTER

SIX

CELLRENDERERSPIN

Using a CellRendererSpin widget provides a *SpinButton* within a TreeView. The widget contains an entry for manually inputting numbers, as well as two buttons to adjust the value contained up or down.

Note: See the *CellRenderer* page for additional methods available.

116.1 Constructor

The CellRendererSpin can be constructed using the following:

```
cellrendererspin = Gtk.CellRendererSpin()
```

116.2 Methods

The method for setting properties on a CellRendererSpin via:

```
cellrendererspin.set_property(property, value)
```

116.3 Properties

The configuration of the CellRendererSpin is made using the property functions:

```
cellrendererspin.set_property("item", value)
```

The property items available for use with the CellRendererSpin are:

- `"digits"` - the number of digits displayed after the decimal point can be specified for values which require it.
- `"adjustment"` - specify the *Adjustment* object from which the current value, hightest and lowest values, and also the incremental value should be taken.

116.4 Example

Below is an example of an CellRendererSpin:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class CellRendererSpin(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(150, -1)
        self.connect("destroy", Gtk.main_quit)

        self.liststore = Gtk.ListStore(str, int)
        self.liststore.append(["Oranges", 5])
        self.liststore.append(["Bananas", 2])
        self.liststore.append(["Apples", 3])

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        adjustment = Gtk.Adjustment(0, 0, 10, 1, 1, 0)
        cellrendererspin = Gtk.CellRendererSpin()
        cellrendererspin.set_property("editable", True)
        cellrendererspin.set_property("adjustment", adjustment)
        cellrendererspin.connect("edited", self.on_cell_edited)

        treeviewcolumn = Gtk.TreeViewColumn("Fruit")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, False)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        treeviewcolumn = Gtk.TreeViewColumn("Quantity")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrendererspin, False)
        treeviewcolumn.add_attribute(cellrendererspin, "text", 1)

    def on_cell_edited(self, cellrendererspin, treepath, value):
        self.liststore[treepath][1] = int(value)

window = CellRendererSpin()
window.show_all()

Gtk.main()
```

Download: `CellRendererSpin`

CHAPTER

SEVEN

CELLRENDERERCOMBO

A CellRendererCombo widget provides a dropdown menu within a *TreeView*. The functionality is similar to that provide by a *ComboBox*.

Note: See the *CellRenderer* page for additional methods available.

117.1 Constructor

The CellRendererCombo can be constructed using the following:

```
cellrenderercombo = Gtk.CellRendererCombo()
```

117.2 Properties

The configuration of the CellRendererCombo is made using the property functions:

```
cellrenderercombo.set_property("item", value)
```

The property items available for use with the CellRendererCombo are:

- `"has-entry"` - setting to `True` enables text to be entered by the user along with selecting a value from a menu.
- `"model"` - specifies the data store which will be used to retrieve values from. This should be set to the name of an appropriate *ListStore* or *TreeStore* containing the values.
- `"text-column"` - used to set the column from which the values are to be retrieved.

117.3 Signals

The common signals of the CellRendererCombo are:

```
"changed" (cellrenderercombo, path, treeiter)
```

The `"changed"` signal emits when the user changes which item has been selected. The *path* value is a string which identifies the cell relative to the tree view model. The *treeiter* is the iter of the item selected relative to the combo box model. These items can be put together to return the item from the model.

117.4 Example

Below is an example of a CellRendererCombo:

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

from gi.repository import Gtk

class CellRendererCombo(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        self.liststoreAppliance = Gtk.ListStore(str, str)
        self.liststoreAppliance.append(["Dishwasher", "Bosch"])
        self.liststoreAppliance.append(["Refrigerator", "Samsung"])
        self.liststoreAppliance.append(["Cooker", "Hotpoint"])

        self.liststoreManufacturers = Gtk.ListStore(str)
        self.liststoreManufacturers.append(["Bosch"])
        self.liststoreManufacturers.append(["Whirlpool"])
        self.liststoreManufacturers.append(["Hotpoint"])
        self.liststoreManufacturers.append(["DeLonghi"])
        self.liststoreManufacturers.append(["Samsung"])

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststoreAppliance)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Appliance")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, False)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        cellrenderercombo = Gtk.CellRendererCombo()
        cellrenderercombo.set_property("editable", True)
        cellrenderercombo.set_property("model", self.liststoreManufacturers)
        cellrenderercombo.set_property("text-column", 0)
        cellrenderercombo.connect("changed", self.on_combo_changed)

        treeviewcolumn = Gtk.TreeViewColumn("Manufacturer")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderercombo, False)
        treeviewcolumn.add_attribute(cellrenderercombo, "text", 1)

    def on_combo_changed(self, cellrenderercombo, treepath, treeiter):
        self.liststoreAppliance[treepath][1] = self.
↪liststoreManufacturers[treeiter][0]

window = CellRendererCombo()
window.show_all()

Gtk.main()
```

Download: `CellRendererCombo`

CHAPTER

EIGHT

CELLRENDERERPROGRESS

A CellRendererProgress allows for the displaying of a *ProgressBar* within a TreeView to show the status, or percentage completion of a task.

Note: See the *CellRenderer* page for additional methods available.

118.1 Constructor

The CellRendererProgress can be constructed using the following:

```
cellrendererprogress = Gtk.CellRendererProgress()
```

118.2 Properties

The configuration of the CellRendererProgress is made using the property functions:

```
cellrendererprogress.set_property("item", value)
```

The property items available for use with the CellRendererProgress are:

- `"pulse"` - when the attribute is set to `True`, the ProgressBar will pulse from side-to-side to indicate activity.
- `"text"` - text content to be displayed within the cell.
- `"value"` - a value specified between 0 and 100 indicating the amount of progress made.
- `"inverted"` - to reverse the direction of the ProgressBar, use `True` as the value.

118.3 Example

Below is an example of an CellRendererProgress:

```
#!/usr/bin/env python3

from gi.repository import Gtk, GObject
import random

```

(continues on next page)

(continued from previous page)

```
class CellRendererProgress(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        self.liststore = Gtk.ListStore(str, int)
        self.liststore.append(["Downloading files", 0])
        self.liststore.append(["Parsing access logs", 0])
        self.liststore.append(["Compiling modules", 0])

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Action")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, False)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        cellrendererprogress = Gtk.CellRendererProgress()

        treeviewcolumn = Gtk.TreeViewColumn("Status")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrendererprogress, True)
        treeviewcolumn.add_attribute(cellrendererprogress, "value", 1)

        GObject.timeout_add(250, self.on_pulse_progressbar)

    def on_pulse_progressbar(self):
        for item in self.liststore:
            if item[1] < 100:
                value = random.randint(0, 5)

                if value + item[1] > 100:
                    item[1] = 100
                else:
                    item[1] += value
            else:
                item[1] = 0

        return True

window = CellRendererProgress()
window.show_all()

Gtk.main()
```

Download: `CellRendererProgress`

CHAPTER

NINE

CELLRENDERERPIXBUF

A CellRendererPixbuf provides a way to display images and icons within a TreeView cell.

Note: See the *CellRenderer* page for additional methods available.

119.1 Constructor

The CellRendererPixbuf can be constructed using the following:

```
cellrendererpixbuf = Gtk.CellRendererPixbuf()
```

119.2 Properties

The configuration of the CellRendererPixbuf is made using the property functions:

```
cellrendererpixbuf.set_property("item", value)
```

The property items available for use with the CellRendererPixbuf are:

- `"pixbuf"` - the pixbuf value sets the location of the Pixbuf image format to display in the cell.

119.3 Example

Below is an example of an CellRendererPixbuf:

```
#!/usr/bin/env python3

from gi.repository import Gtk, GdkPixbuf

class CellRendererPixbuf(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        liststore = Gtk.ListStore(str, GdkPixbuf.Pixbuf)
```

(continues on next page)

(continued from previous page)

```
        icon = GdkPixbuf.Pixbuf.new_from_file_at_size("../_resources/fedora.ico", 16,␣
↪16)
        liststore.append(["Fedora", icon])
        icon = GdkPixbuf.Pixbuf.new_from_file_at_size("../_resources/opensuse.ico",␣
↪16, 16)
        liststore.append(["OpenSuSE", icon])
        icon = GdkPixbuf.Pixbuf.new_from_file_at_size("../_resources/gentoo.ico", 16,␣
↪16)
        liststore.append(["Gentoo", icon])

        treeview = Gtk.TreeView()
        treeview.set_model(liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Distribution")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        cellrendererpixbuf = Gtk.CellRendererPixbuf()

        treeviewcolumn = Gtk.TreeViewColumn("Logo")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrendererpixbuf, False)
        treeviewcolumn.add_attribute(cellrendererpixbuf, "pixbuf", 1)

window = CellRendererPixbuf()
window.show_all()

Gtk.main()
```

Download: `CellRendererPixbuf`

CHAPTER

CELLRENDERERACCEL

A CellRendererAccel is used to render a keyboard accelerator into a cell. The object is editable if required and allows a user to change the accelerator by entering a new combination.

Note: See the *CellRenderer* page for additional methods available.

120.1 Constructor

The CellRendererAccel can be constructed using the following:

```
cellrendereraccel = Gtk.CellRendererAccel()
```

120.2 Properties

The configuration of the CellRendererAccel is made using the property functions:

```
cellrendereraccel.set_property("item", value)
```

The property items available for use with the CellRendererAccel are:

- `"editable"` - when specified as `True`, the accelerator value can be edited by the user.

120.3 Signals

The commonly used signals of a CellRendererAccel are:

```
"accel-cleared" (cellrendereraccel, path_string)
"accel-edited" (cellrendereraccel, path_string, accel_key, accel_mods, hardware_
↪keycode)
```

The `"accel-cleared"` signals is emitted when the user presses the `Backspace` key. On the other hand, `"accel-edited"` is seen when the user changes the accelerator by entering another key combination.

120.4 Example

Below is an example of an CellRendererAccel:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class CellRendererAccel(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        self.liststore = Gtk.ListStore(str, str)
        self.liststore.append(["New", "<Primary>n"])
        self.liststore.append(["Open", "<Primary>o"])
        self.liststore.append(["Save", "<Primary>s"])

        treeview = Gtk.TreeView()
        treeview.set_model(self.liststore)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()

        treeviewcolumn = Gtk.TreeViewColumn("Action")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrenderertext, True)
        treeviewcolumn.add_attribute(cellrenderertext, "text", 0)

        cellrendereraccel = Gtk.CellRendererAccel()
        cellrendereraccel.set_property("editable", True)
        cellrendereraccel.connect("accel-edited", self.on_accel_edited)
        cellrendereraccel.connect("accel-cleared", self.on_accel_cleared)

        treeviewcolumn = Gtk.TreeViewColumn("Accelerator")
        treeview.append_column(treeviewcolumn)
        treeviewcolumn.pack_start(cellrendereraccel, True)
        treeviewcolumn.add_attribute(cellrendereraccel, "text", 1)

    def on_accel_edited(self, cellrendereraccel, path, key, mods, hwcode):
        accelerator = Gtk.accelerator_name(key, mods)
        self.liststore[path][1] = accelerator

    def on_accel_cleared(self, cellrendereraccel, path):
        self.liststore[path][1] = "None"

window = CellRendererAccel()
window.show_all()

Gtk.main()
```

Download: `CellRendererAccel`

CHAPTER

ONE

TREEMODELSORT

A TreeModelSort object allows the data within a *ListStore* or *TreeStore* to be sorted in a variety of different ways. It can be seen as a layer which sits between the model and the viewing widget, such as a *TreeView* or *ComboBox* as it sorts the data passed.

121.1 Constructor

Constructing the TreeModelSort object is done with:

```
treemodelsort = Gtk.TreeModelSort(model)
```

The *model* parameter is the name of the TreeStore or ListStore which is going to be sorted.

121.2 Methods

If required, the model attached to the TreeModelSort can be retrieved via:

```
treemodelsort.get_model()
```

Basic sorting is achievable by using:

```
treemodelsort.set_sort_column_id(column, sort_type)
```

The *column* value is the number of the column to be sorted, with the first column being represented by `0`. The *sort_type* paraemter is a Gtk constant which indicates the direction of the sorting, and should be set to either `Gtk.SortType.ASCENDING` for A-Z, or `Gtk.SortType.DESCENDING` for Z-A.

Custom sorting, used typically for the ability of sorting by multiple columns is also available at the cost of being more complex. It requires a function to be defined where the sorting comparison takes place, with the return values from the function indicating to GTK+ whether the item will go above or below the previous one. The custom sorting method is:

```
treemodelsort.set_sort_func(column, function, data)
```

The *column* value specified is the column where the sort function to be applied. The *function* parameter is the name of the function where the sorting comparison will be done. Specifying the *data* parameter can also be done which provides data to be sorted, or alternatively can be left out entirely.

As the TreeModelSort sits between the viewing widget and data model, the correct TreeIter or TreePath for the underlying model is not returned. These need to be converted using:

```
treemodelsort.convert_iter_to_child_iter(treeiter)
treemodelsort.convert_path_to_child_path(treepath)
```

Both functions return the TreeIter or TreePath for the unsorted model.

121.3 Example

Below is an example of a TreeModelSort:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class TreeModelSort(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, -1)
        self.connect("destroy", Gtk.main_quit)

        liststore = Gtk.ListStore(str)
        liststore.append(["Mark"])
        liststore.append(["Chris"])
        liststore.append(["Tim"])
        liststore.append(["David"])
        liststore.append(["Keith"])

        treemodelsort = Gtk.TreeModelSort(liststore)
        treemodelsort.set_sort_column_id(0, Gtk.SortType.ASCENDING)

        treeview = Gtk.TreeView()
        treeview.set_model(treemodelsort)
        self.add(treeview)

        cellrenderertext = Gtk.CellRendererText()
        treeviewcolumn = Gtk.TreeViewColumn("Name", cellrenderertext, text=0)
        treeview.append_column(treeviewcolumn)

window = TreeModelSort()
window.show_all()

Gtk.main()
```

Download: `TreeModelSort`

CHAPTER

TWO

TREEMODELFILTER

The TreeModelFilter objects provides a way to filter the data displayed in a *ListStore* or *TreeStore* via a function, which returns whether the data is shown or not.

122.1 Constructor

The TreeModelFilter is typically constructed from an existing model:

```
treemodelfilter = model.filter_new()
```

It can however be constructed itself, with the model being derived from the filter:

```
treemodelfilter = Gtk.TreeModelFilter()
```

122.2 Methods

A function is attached to the TreeModelFilter which returns `True` or `False` depending on whether the row is to be displayed or not. The function is called for each row in the model, and when `False` is returned, the row is NOT displayed. The function is set via:

```
treemodelfilter.set_visible_func(function, data)
```

The *data* specifies the data which is being displayed in the TreeModelFilter.

The TreeModelFilter can be refiltered using the method:

```
treemodelfilter.refilter()
```

In some cases, it may be useful to use columns within the data model which indicate whether a row should be displayed. Setting the column type to a Boolean type, the column can be declared with:

```
treemodelfilter.set_visible_column(column)
```

The *column* value should be set to the number of the column containing the visibility information. When the row is set to `True` it is displayed.

When retrieving the selected item from the viewing widget, it will return a TreeIter or TreePath for the filter. These need to be converted to correctly access the underlying model:

```
treemodelfilter.convert_iter_to_child_iter(treeiter)
treemodelfilter.convert_path_to_child_path(treepath)
```

A convenience function can be used to return the model being filtered:

```
treemodefilter.get_model()
```

122.3 Example

Below is an example of a TreeModelFilter:

```
#!/usr/bin/env python3

from gi.repository import Gtk

products = (("Apple", "Fruit", "£0.20"),
            ("Bleach", "Cleaning", "£1.20"),
            ("Bird Seed", "Pets", "£2.50"),
            ("Banana", "Fruit", "£0.35"),
            ("Beer", "Alcohol", "£2.75"),
            ("Cornflakes", "Cereal", "£1.10"),
            ("Pineapple", "Fruit", "£0.75"),
           )

class TreeModelFilter(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        grid = Gtk.Grid()
        grid.set_row_spacing(5)
        self.add(grid)

        scrolledwindow = Gtk.ScrolledWindow()
        scrolledwindow.set_vexpand(True)
        scrolledwindow.set_policy(Gtk.PolicyType.NEVER, Gtk.PolicyType.NEVER)
        grid.attach(scrolledwindow, 0, 0, 1, 1)

        liststore = Gtk.ListStore(str, str, str)
        self.treemodelfilter = liststore.filter_new()
        self.treemodelfilter.set_visible_func(self.filter_visible, products)

        self.combobox = Gtk.ComboBoxText()
        self.combobox.append_text("All")
        self.combobox.set_active(0)
        self.combobox.connect("changed", self.on_category_changed)
        grid.attach(self.combobox, 0, 1, 1, 1)

        for product in products:
            liststore.append(product)

            self.combobox.append_text(product[1])

        treeview = Gtk.TreeView()
        treeview.set_model(self.treemodelfilter)
        scrolledwindow.add(treeview)

        cellrenderertext = Gtk.CellRendererText()
```

(continues on next page)

(continued from previous page)

```
        treeviewcolumn = Gtk.TreeViewColumn("Product", cellrenderertext, text=0)
        treeview.append_column(treeviewcolumn)
        treeviewcolumn = Gtk.TreeViewColumn("Category", cellrenderertext, text=1)
        treeview.append_column(treeviewcolumn)
        treeviewcolumn = Gtk.TreeViewColumn("Price", cellrenderertext, text=2)
        treeview.append_column(treeviewcolumn)

    def on_category_changed(self, combobox):
        self.treemodelfilter.refilter()

    def filter_visible(self, model, treeiter, data):
        show = False

        if model[treeiter][1] == self.combobox.get_active_text():
            show = True
        elif self.combobox.get_active_text() == "All":
            show = True

        return show

window = TreeModelFilter()
window.show_all()

Gtk.main()
```

Download: `TreeModelFilter`

CHAPTER

THREE

ICONVIEW

The purpose of the IconView is to displays images in thumbnail format, arranged into a grid format. Items can be added with optional text labels and tooltips, with the ability to customise selections if required.

123.1 Constructor

The IconView can be constructed using the following:

```
iconview = Gtk.IconView(model)
```

The *model* value is optional, however allows a data store to be attached to the IconView at construction.

123.2 Methods

To set the model after constructing the widget call:

```
iconview.set_model(model)
```

The *model* parameter should be set to a *ListStore* which is used to hold the associated images, text or tooltip information.

To set the image, text and tooltip columns from the IconView the following calls can be made:

```
iconview.set_pixbuf_column(column)
iconview.set_text_column(column)
iconview.set_markup_column(column)
iconview.set_tooltip_column(column)
```

The *column* parameter indicates the column number within the ListStore, with `0` identifying the first column.

To set the width of the image within the IconView use:

```
iconview.set_item_width(item_width)
```

The *item_width* should be specified as an integer with the value specifying the number of pixels in width each image is.

A number of permitted columns can also be defined using:

```
iconview.set_columns(columns)
```

If *columns* is set to `-1`, the number will be determined to fill the space available.

The spacing defined between the icon and label text can be set in pixels by:

```
iconview.set_spacing(spacing)
```

Row and column spacing can also be defined separately with:

```
iconview.set_row_spacing(spacing)
iconview.set_column_spacing(spacing)
```

A margin value can also be set to provide a gap between the IconView frame and content:

```
iconview.set_margin(margin)
```

Padding of an item can also be defined with the method:

```
iconview.set_item_padding(padding)
```

To allow items within the IconView to be reordered use:

```
iconview.set_reorderable(reorderable)
```

When the *reorderable* parameter is set to `True`, images can be dragged-and-dropped into a new position.

By default, all text names are shown below the image, however this can be configured using:

```
iconview.set_item_orientation(orientation)
```

The *orientation* parameter should be set to one of the following; `Gtk.Orientation.HORIZONTAL` or `Gtk.Orientation.VERTICAL`. When `Gtk.Orientation.HORIZONTAL` is used, text is displayed to the right of the image.

The default setting of the IconView prevents more than one item being selected at any one time, however this can be changed with:

```
iconview.set_selection_mode(selection_mode)
```

The *selection_mode* argument can be set to either `Gtk.SelectionMode.NONE` which prevents any selection, `Gtk.SelectionMode.SINGLE` that allows only a single item to be selected, `Gtk.SelectionMode.BROWSE` allows one selection however at default can allow multiple if required, or `Gtk.SelectionMode.MULTIPLE` which allows the user to hold down the `Control` key to highlight multiple items.

By default, items must be double-clicked to activated. This can be changed to a single-click by using:

```
iconview.set_activate_on_single_click(single_click)
```

When *single_click* is set to `True`, selecting the item will activate the item.

All items within the IconView can be selected or unselected with:

```
iconview.select_all()
iconview.unselect_all()
```

A list of selected items within the IconView can be obtained by calling:

```
iconview.get_selected_items()
```

The list returned will contain TreePath objects for each item selected.

123.3 Signals

The commonly used signals of an IconView are:

```
"item-activated" (iconview, path)
"selection-changed" (iconview)
"select-all" (iconview)
"unselect-all" (iconview)
"move-cursor" (iconview, step, count)
```

An "item-activated" signal emits from the IconView widget when the user double-clicks an item or when Return/Enter is pressed. The IconView parameter is passed along with a TreePath identifying the location of the item. The "selection-changed" is emitted when the highlighted item is changed. An event of "select-all" or "unselect-all" occurs when all or none of the items are selected. All three signals emit only the iconview on which the event occurred. Finally, the "move-cursor" signal occurs when the user initiates a cursor movement. It passes the iconview on which the event run, the step which specifies the type of movement and the count which identifies the number of units the move made.

123.4 Example

Below is an exmaple of a IconView:

```
#!/usr/bin/env python3

from gi.repository import Gtk
from gi.repository import GdkPixbuf

distributions = ["fedora", "mandriva", "zenwalk", "knoppix", "debian"]

class IconView(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        self.liststore = Gtk.ListStore(str, GdkPixbuf.Pixbuf, str)

        iconview = Gtk.IconView()
        iconview.set_model(self.liststore)
        iconview.set_text_column(0)
        iconview.set_pixbuf_column(1)
        iconview.set_tooltip_column(2)
        iconview.connect("item-activated", self.on_iconview_activated)
        self.add(iconview)

        image = Gtk.Image()

        for item in distributions:
            path = "../_resources/%s.ico" % (item)
            image.set_from_file(path)
            pixbuf = image.get_pixbuf()

            name = item.capitalize()
            tooltip = "%s tooltip example" % (name)
```

(continues on next page)

(continued from previous page)

```
            self.liststore.append([name, pixbuf, tooltip])

    def on_iconview_activated(self, iconview, treepath):
        print("Selected item: %s" % (self.liststore[treepath][0]))

window = IconView()
window.show_all()

Gtk.main()
```

Download: `IconView`

CHAPTER

FOUR

LISTBOX

The ListBox widget provides a vertical container to display any widgets which are inserted into it. The widget also provides functions for sorting and filtering. It is an alternative to a *TreeView* in some cases, and is suitable for inserting a wide range of widgets quickly.

124.1 Constructor

The ListBox can be constructed using:

```
listbox = Gtk.ListBox()
```

124.2 Methods

Rows can be inserted into the ListBox using the following statements:

```
listbox.add(child)
listbox.insert(child, position)
listbox.prepend(child)
```

The selection mode of the child widgets in the ListBox can be configured:

```
listbox.set_selection_mode(selection_mode)
```

The *selection_mode* parameter can be set to one of the following:

- `Gtk.Selection.NONE`
- `Gtk.Selection.SINGLE`
- `Gtk.Selection.BROWSE`

When there are no child widgets to display in the ListBox, a placeholder can be set instead. This could be a *Label* or *Image* for example.

listbox.set_placeholder(child)

By default, the ListBoxRow is activated when the user single clicks, however in some cases it may be useful to require a double click.

listbox.set_activate_on_single_click(single_click)

When *single_click* is set to `False`, the user requires a double-click to activate the row. This also allows the `"row-selected"` signal to work.

124.3 Signals

The commonly use signals of a ListBox are:

```
"row-activated" (listbox, listboxrow)
"row-selected" (listbox, listboxrow)
```

The `"row-activated"` signal emits when the user single clicks on the ListBoxRow. If the method `.activate_on_single_click()` is set to False, the user must double click. When single click activate is False, the `"row-selected"` signal can then be used, and emits when the user highlights a row.

124.4 Example

Below is an example of a FlowBox:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class FlowBox(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        flowbox = Gtk.FlowBox()
        flowbox.connect("child-activated", self.on_child_activated)
        self.add(flowbox)

        for count in range(0, 9):
            label = Gtk.Label("Row %i" % (count))
            flowbox.add(label)

    def on_child_activated(self, flowbox, flowboxchild):
        print("Child %i activated" % (flowboxchild.get_index()))

window = FlowBox()
window.show_all()

Gtk.main()
```

Download: `FlowBox`

CHAPTER

FIVE

FLOWBOX

The FlowBox is a container which allows automatic reflowing of child widgets, according to their orientation. When the FlowBox is set to horizontal orientation, widgets are arranged from left to right, and new rows are started when necessary. Vertical orientation arranged children from top to bottom and create a new column when necessary.

The widget is similar to a *ListBox*.

125.1 Constructor

The FlowBox can be constructed using:

```
flowbox = Gtk.FlowBox()
```

125.2 Methods

Widgets are inserted into the FlowBox via the method:

```
flowbox.insert(child, position)
```

The *child* parameter will typically be another container such as *Box* or *Grid*. Widgets can be inserted however. The *position* value is an integer value, starting at `0` which indicates the first position.

Spacing between the children contained in the FlowBox can be set for both column and row:

```
flowbox.set_row_spacing(spacing)
flowbox.set_column_spacing(spacing)
```

To force all child widgets to be the same size regardless of the size they require:

```
flowbox.set_homogeneous(homogeneous)
```

Items within the FlowBox can be selected or unselected:

```
flowbox.select_all()
flowbox.unselect_all()
```

Typically, the FlowBox places child widgets automatically. This can lead to an undesirable number in a single row or column. Minimum and maximum numbers can therefore be set:

```
flowbox.set_min_children_per_line(minimum)
flowbox.set_max_children_per_line(maximum)
```

Controlling the number of clicks it takes to activate an item is configured with:

```
flowbox.set_activate_on_single_click(single)
```

When *single* is set to `True`, an item is activated on a single click. When using `False`, a double-click is required.

The number of items in the FlowBox which can be selected is configurable via:

```
flowbox.set_selection_mode(mode)
```

The *mode* value can be set to:

- `Gtk.Selection.NONE` - prevent any selection being made.
- `Gtk.Selection.SINGLE` - allow only a single selected item.
- `Gtk.Selection.BROWSE` - allow one or more selected items.
- `Gtk.Selection.MULTIPLE` - allow multiple selected items.

125.3 Signals

The commonly used signals of a FlowBox are:

```
"activate" (child)
"select-all" (box)
"unselect-all" (box)
```

The `"activate"` signal is emitted when the user either single-clicks or double-clicks on a child. The FlowBox emits the `"select-all"` and `"unselect-all"` signals when all the items are chosen or unchosen.

125.4 Example

Below is an example of a FlowBox:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class FlowBox(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_default_size(200, 200)
        self.connect("destroy", Gtk.main_quit)

        flowbox = Gtk.FlowBox()
        flowbox.connect("child-activated", self.on_child_activated)
        self.add(flowbox)

        for count in range(0, 9):
            label = Gtk.Label("Row %i" % (count))
            flowbox.add(label)

    def on_child_activated(self, flowbox, flowboxchild):
        print("Child %i activated" % (flowboxchild.get_index()))
```

(continues on next page)

(continued from previous page)

```
window = FlowBox()
window.show_all()

Gtk.main()
```

Download: `FlowBox`

CHAPTER

SIX

DIALOG

Dialog widgets can be used to prompt or request information from the user. They are commonly used for preference dialogs where a user can configure an application.

126.1 Constructor

The Dialog can be constructed using the following:

```
dialog = Gtk.Dialog()
```

126.2 Methods

Once the Dialog has been constructed, it can be displayed, and then subsequently destroyed with:

```
dialog.run()
dialog.destroy()
```

When `dialog.run()` is called, GTK+ loops until it receives a response. The response can be clicking on the X of the Dialog window, or clicking one of the Button's. Once the Dialog receives a response, `dialog.destroy` is then called.

Note: If you application only uses a Dialog, the `Gtk.main()` call is not required. This is invoked automatically when calling `dialog.run()`.

A Dialog widget comes pre-packed with a vertically oriented *Box* container from which additional widgets can be added. This is retrieved using:

```
dialog.get_content_area()
```

The Box can then be added to or manipulated using the methods found on the *Box* page.

To set the default response of the Dialog, the following method is usable:

```
dialog.set_default_response(response)
```

The *response_id* can be set to any of the following constant values:

- `Gtk.ResponseType.NONE`
- `Gtk.ResponseType.OK`

- `Gtk.ResponseType.CANCEL`
- `Gtk.ResponseType.YES`
- `Gtk.ResponseType.NO`
- `Gtk.ResponseType.CLOSE`
- `Gtk.ResponseType.ACCEPT`
- `Gtk.ResponseType.REJECT`
- `Gtk.ResponseType.DELETE_EVENT`
- `Gtk.ResponseType.APPLY`
- `Gtk.ResponseType.HELP`.

Alternatively, an integer value can be specified in place of a constant if none of the provided values are suitable.

In some cases, it may be necessary for one of the Dialog buttons to be as not sensitive (greyed-out). This can be set by calling:

```
dialog.set_response_sensitive(response, setting)
```

The *response* parameter should be an appropriate response type, with the selection being from those listed above. The *setting* argument should then be a Boolean value, with `False` setting the button to insensitive.

By default, the Dialog contains no buttons. These can be added with:

```
dialog.add_button(label, response)
```

The *label* should be set to the text which is to be displayed on the Button. A *response* should be appropriate for the Button type and be one of those listed above.

Alternatively, multiple buttons can be added via a single method using:

```
dialog.add_buttons(label, response, ...)
```

The *label* parameter should be a textual string which will be displayed to the user. The *response* should be an integer value identifying the response which the button causes.

The area of the Dialog containing the buttons is known as the Action area. Other widgets can be added to this area using:

```
dialog.add_action_widget(child, response_id)
```

The *child* should be set to the widget which is to be added. The *response* determines the output of the child widget when it is triggered.

To find a response which matches a widget, or a widget that matches a response use the methods:

```
dialog.get_widget_for_response(response)
dialog.get_response_for_widget(widget)
```

Setting the title of the dialog after construction is possible by calling:

```
dialog.set_title(title)
```

A parent should be defined for the Dialog by calling:

```
dialog.set_transient_for(parent)
```

Note: If not transient (parent) window is defined, GTK+ will display a warning message that a parent should be defined. When a parent window is defined, the dialog is centered in the center of the parent window, and is destroyed when the parent is destroyed.

126.3 Signals

The commonly used signals of a Dialog are:

```
"close" (dialog)
"response" (dialog, response)
```

The `"close"` event occurs when the user presses the `Escape` button on the keyboard, or the `Gtk.ResponseType.CLOSE` response is met. Alternatively, `"response"` can be emitted when anything happens within the Dialog. Both events emit the Dialog object with the function, however the `"response"` signal also emits a response_id value of the event that occurred within the Dialog.

126.4 Examples

Below is an example of a Dialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Dialog(Gtk.Dialog):
    def __init__(self):
        Gtk.Dialog.__init__(self)
        self.set_title("Dialog")
        self.set_default_size(400, 300)
        self.add_button("_OK", Gtk.ResponseType.OK)
        self.add_button("_Cancel", Gtk.ResponseType.CANCEL)
        self.connect("response", self.on_response)

        label = Gtk.Label("This is a Dialog example.")
        self.vbox.add(label)

        self.show_all()

    def on_response(self, dialog, response):
        if response == Gtk.ResponseType.OK:
            print("OK button clicked")
        elif response == Gtk.ResponseType.CANCEL:
            print("Cancel button clicked")
        else:
            print("Dialog closed")

        dialog.destroy()

dialog = Dialog()
dialog.run()
```

Download: `Dialog`

CHAPTER

SEVEN

MESSAGEDIALOG

A MessageDialog is used to display information or ask questions of the user. These messages are displayed within the Window that the user is working.

The MessageDialog is similar in use case to the *InfoBar* widget.

127.1 Constructor

The MessageDialog can be constructed using the following:

```
messagedialog = Gtk.MessageDialog(message_type, message_format)
```

The *message_type* indicates the type of message which the MessageDialog will be displayed for. Setting an appropriate message type for the MessageDialog is important to ensure that the user quickly understands the context of the message. The options for this value are:

- `Gtk.MessageType.INFO` - used to display information messages.
- `Gtk.MessageType.WARNING` - used to display warning messages.
- `Gtk.MessageType.QUESTION` - used to display question messages.
- `Gtk.MessageType.ERROR` - used to display error messages.
- `Gtk.MessageType.OTHER` - used to display other messages.

The *message_format* should be set to the text which is to be displayed within the MessageDialog.

127.2 Methods

After the MessageDialog has been constructed, it can be run and destroyed with:

```
messagedialog.run()
messagedialog.destroy()
```

GTK+ will loop in the `.run()` method until it receives a response, upon which any code that needs to be run is executed (for example, responding to the users request). After completion, the `.destroy()` method will remove the MessageDialog.

Normal or markup-based text can be added to the MessageDialog via:

```
messagedialog.set_markup(text)
```

By default a MessageDialog only has one line of text. To add a second level of text or markup use:

```
messagedialog.format_secondary_text(text)
messagedialog.format_secondary_markup(markup)
```

When secondary text is in use, the primary text entered at construction time is made bold and enlarged. The secondary text then takes the place of the primary text. The use case for this is to provide a quick overview with the primary, and a further explanation with the secondary.

The title of the MessageDialog can be set after construction via:

```
messagedialog.set_title(title)
```

A MessageDialog should also be attached to a parent:

```
messagedialog.set_transient_for(parent)
```

The *parent* value is the name of the parent window which called the MessageDialog.

Note: If no transient (parent) window is defined, GTK+ will display a warning message that a parent should be defined. When a parent window is defined, the dialog is centered in the center of the parent window, and is destroyed when the parent is destroyed.

Buttons are attached to the MessageDialog with the method:

```
messagedialog.add_button(label, response)
```

The *button* value should be set to a string of text identifying the function of the button. The *response* indicates the response the button emits.

Alternatively, multiple buttons can be added to the MessageDialog in a single method:

```
messagedialog.add_buttons(label, response, label, response, ...)
```

A default button can be selected on the MessageDialog with:

```
messagedialog.set_default_response(response)
```

The *response* type must be defined via the added buttons, otherwise it will be ignored. It should be set appropriately such that the user does not lose any data (e.g. a delete operation), but is favourable to the purpose (e.g. a continue operation).

A response can be set sensitive if required with the method:

```
messagedialog.set_response_sensitive(response, sensitive)
```

When the *sensitive* parameter is set to `False`, the defined *response* will be "greyed-out". This is not typically seen on a MessageDialog except when an additional widget is included which must be handled prior to continuing.

127.3 Properties

The message type can be set with the property:

```
messagedialog.set_property("message-type", message_type)
```

The *message_type* can be set to `Gtk.MessageType.INFO`, `Gtk.MessageType.WARNING`, `Gtk.MessageType.QUESTION`, `Gtk.MessageType.ERROR`, or `Gtk.MessageType.OTHER` depending on the desired output.

The main text of the dialog can be configured using:

```
messagedialog.set_property("text", text)
```

127.4 Example

Below is an example of a MessageDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class MessageDialog(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.set_title("MessageDialog")
        self.connect("destroy", Gtk.main_quit)

        buttonbox = Gtk.ButtonBox()
        self.add(buttonbox)

        buttonInformation = Gtk.Button(label="Information")
        buttonInformation.message_type = Gtk.MessageType.INFO
        buttonInformation.connect("clicked", self.on_message_clicked)
        buttonbox.add(buttonInformation)
        buttonWarning = Gtk.Button(label="Warning")
        buttonWarning.message_type = Gtk.MessageType.WARNING
        buttonWarning.connect("clicked", self.on_message_clicked)
        buttonbox.add(buttonWarning)
        buttonQuestion = Gtk.Button(label="Question")
        buttonQuestion.message_type = Gtk.MessageType.QUESTION
        buttonQuestion.connect("clicked", self.on_message_clicked)
        buttonbox.add(buttonQuestion)
        buttonError = Gtk.Button(label="Error")
        buttonError.message_type = Gtk.MessageType.ERROR
        buttonError.connect("clicked", self.on_message_clicked)
        buttonbox.add(buttonError)
        buttonOther = Gtk.Button(label="Other")
        buttonOther.message_type = Gtk.MessageType.OTHER
        buttonOther.connect("clicked", self.on_message_clicked)
        buttonbox.add(buttonOther)

        self.messagedialog = Gtk.MessageDialog(message_format="MessageDialog")
        self.messagedialog.set_transient_for(self)
        self.messagedialog.set_title("MessageDialog")
        self.messagedialog.set_markup("<span size='12000'><b>This is a MessageDialog␣
↪widget.</b></span>")
        self.messagedialog.format_secondary_text("The MessageDialog can display a␣
↪main message, and further secondary content.")
        self.messagedialog.add_button("_Close", Gtk.ResponseType.CLOSE)

    def on_message_clicked(self, button):
```

(continues on next page)

(continued from previous page)

```
        self.messagedialog.set_property("message-type", button.message_type)

        self.messagedialog.run()
        self.messagedialog.hide()

window = MessageDialog()
window.show_all()

Gtk.main()
```

Download: `MessageDialog`

CHAPTER

EIGHT

ABOUTDIALOG

The AboutDialog is found in virtually all non-trivial applications. It provides access to the version number, license, authors, translators and other information relating to the development of the program.

128.1 Constructor

The AboutDialog can be constructed using:

```
aboutdialog = Gtk.AboutDialog()
```

128.2 Methods

Once constructed, the AboutDialog will be void of information with the exception of the name of the Python script. A variety of information can now be added with:

```
aboutdialog.set_program_name(name)
aboutdialog.set_version(version)
aboutdialog.set_copyright(copyright)
aboutdialog.set_comments(comment)
aboutdialog.set_website(website)
aboutdialog.set_website_label(label)
aboutdialog.set_authors([authors])
aboutdialog.set_documenters([documenters])
aboutdialog.set_artists([artists])
aboutdialog.set_logo(filepath)
aboutdialog.set_wrap_license(wrap)
```

Most of the options are self-explanatory and take a string of text. The `.set_comments()` method is a simple message about what the application is or does. The `.set_wrap_license()` method tells the license text to be displayed neatly within the dialog window, and should in most cases will be set to `True`. The `.set_authors()`, `.set_documenters()` and `.set_artists()` methods take a Python list of values, with each persons name separated by a comma and contained within square brackets.

The license used by the program can be set with:

```
aboutdialog.set_license_type(license)
```

The *license* value can be set to a number of built-in license types:

- `Gtk.License.UNKNOWN`

- `Gtk.License.CUSTOM`
- `Gtk.License.GPL_2_0`
- `Gtk.License.GPL_3_0`
- `Gtk.License.LGPL_2_1`
- `Gtk.License.LGPL_3_0`
- `Gtk.License.BSD`
- `Gtk.License.MIT_X11`
- `Gtk.License.ARTISTIC`
- `Gtk.License.GPL_2_0_ONLY`
- `Gtk.License.GPL_3_0_ONLY`
- `Gtk.License.LGPL_2_1_ONLY`
- `Gtk.License.LGPL_3_0_ONLY`

A new section for credits can be added with:

```
aboutdialog.add_credit_section(name, people, people...)
```

The *name* value determines the name of the section in the display. One or multiple *people* values can be added to each section.

To display, and then subsequently remove the AboutDialog from view use:

```
aboutdialog.run()
aboutdialog.destroy()
```

When the `.run()` method is executed, the AboutDialog enters a loop and will continue to be displayed until the AboutDialog is destroyed. The event to destroy occurs when the X button or Close button is pressed.

Note: If you application only uses a Dialog, the `Gtk.main()` call is not required. This is invoked automatically when calling `aboutdialog.run()`.

By default, the AboutDialog positions itself in the center of the screen rather than the center of the parent window, if there is one. To ensure neatness and position the AboutDialog over the window:

```
aboutdialog.set_transient_for(window)
```

The *window* argument should be the name of the parent window.

Note: If not transient (parent) window is defined, GTK+ will display a warning message that a parent should be defined. When a parent window is defined, the dialog is centered in the center of the parent window, and is destroyed when the parent is destroyed.

128.3 Signals

The commonly used signals of an AboutDialog are:

```
"activate-link" (aboutdialog, uri)
```

The "activate-link" signal emits when the user clicks the website button within the AboutDialog. This allows configuring of what happens when the user clicks this link. If "activate-link" is not used, the default action is to open the default web browser on the system and navigate to the website.

128.4 Example

Below is an example of an AboutDialog:

```
#!/usr/bin/env python3

from gi.repository import Gtk
from gi.repository import GdkPixbuf

class AboutDialog(Gtk.AboutDialog):
    def __init__(self):
        logo = GdkPixbuf.Pixbuf.new_from_file_at_size("../_resources/gtk.png", 64, 64)

        Gtk.AboutDialog.__init__(self)
        self.set_title("AboutDialog")
        self.set_name("Programmica")
        self.set_version("1.0")
        self.set_comments("Programming, system and network administration resources")
        self.set_website("https://programmica.com/")
        self.set_website_label("Programmica Website")
        self.set_authors(["Andrew Steele"])
        self.set_logo(logo)
        self.connect("response", self.on_response)

    def on_response(self, dialog, response):
        self.destroy()

aboutdialog = AboutDialog()
aboutdialog.run()
```

Download: AboutDialog

CHAPTER

NINE

ASSISTANT

An Assistant widget provides a preset dialog which allows moving through pages in a specific order. It is commonly used to retrieve configuration settings for complex applications.

129.1 Constructor

The Assistant can be constructed using the following:

```
assistant = Gtk.Assistant()
```

129.2 Methods

There are three methods which can be used to add pages to the widget:

```
assistant.insert_page(widget, page)
assistant.append_page(widget)
assistant.prepend_page(widget)
```

The `.insert()` method requires a *widget* to be specified which is added to the page. This is commonly a Grid or Box container. The *page* value should be an integer indicating the position at which the page is to be inserted. The `.append()` and `.prepend()` methods require just a *widget* to be specified, and are added in the ordered they are called.

Pages can also be removed by calling:

```
assistant.remove_page(page)
```

The *page* argument should be the number of the page which is to be removed.

When pages have been added to the Assistant, it is necessary to provide the type of page via:

```
assistant.set_page_type(widget, page_type)
```

The *widget* parameter is the same widget which was specified when calling the `.insert_page()`, `.append()`, or `.prepend()` methods. The *page_type* constant can be one of:

- `Gtk.AssistantPageType.INTRO`
- `Gtk.AssistantPageType.CONTENT`
- `Gtk.AssistantPageType.PROGRESS`

- `Gtk.AssistantPageType.CONFIRM`
- `Gtk.AssistantPageType.SUMMARY`

A title should also be set on the page to describe help identify the content:

```
assistant.set_page_title(widget, page_title)
```

Again, the *widget* parameter must be the same as the one specified when adding the page. The *page_title* is simply a textual string.

To retrieve the current page being viewed in the Assistant call:

```
page = assistant.get_current_page()
```

Alternatively, to set the page in view use:

```
assistant.set_current_page(page)
```

The *page* parameter should be set to an integer, with `0` indicating the first page within the Assistant.

To return the number of pages in the Assistant run:

```
n_pages = assistant.get_n_pages()
```

A page title can be added to the header with the method:

```
assistant.set_page_title(title)
```

The area of the Assistant where the buttons are located is known as the action area. Extra buttons can be added and removed from the area area using:

```
assistant.add_action_widget(child)
assistant.remove_action_widget(child)
```

Pages can be switched programmatically using the methods:

```
assistant.next_page()
assistant.previous_page()
```

Padding can be applied to the Assistant page via the method:

```
assistant.set_page_has_padding(page, padding)
```

The *page* parameter should be set to the widget being displayed on the page. The *padding* value when set to `True` inserts a space around the edge to clean up the display.

129.3 Signals

The commonly used signals of an Assistant are:

```
"apply" (assistant)
"closed" (assistant)
"cancel" (assistant)
```

The `"apply"` signal emits when the user presses the Apply button on a page. A `"closed"` signal is emitted when the user explicitly closes the Assistant by pressing a Close button. The `"cancel"` signal emits when the user presses the Cancel button, or the X on the titlebar of the Assistant window.

129.4 Example

Below is an example of an Assistant:

```
#!/usr/bin/env python3

from gi.repository import Gtk

class Assistant(Gtk.Assistant):
    def __init__(self):
        Gtk.Assistant.__init__(self)
        self.set_title("Assistant")
        self.set_default_size(400, -1)
        self.connect("cancel", self.on_cancel_clicked)
        self.connect("close", self.on_close_clicked)
        self.connect("apply", self.on_apply_clicked)

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        self.append_page(box)
        self.set_page_type(box, Gtk.AssistantPageType.INTRO)
        self.set_page_title(box, "Page 1: Introduction")
        label = Gtk.Label(label="An 'Intro' page is the first page of an Assistant.␣
↪It is used to provide information about what configuration settings need to be␣
↪configured. The introduction page only has a 'Continue' button.")
        label.set_line_wrap(True)
        box.pack_start(label, True, True, 0)
        self.set_page_complete(box, True)

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        self.append_page(box)
        self.set_page_type(box, Gtk.AssistantPageType.CONTENT)
        self.set_page_title(box, "Page 2: Content")
        label = Gtk.Label(label="The 'Content' page provides a place where widgets␣
↪can be positioned. This allows the user to configure a variety of options as needed.
↪ The page contains a 'Continue' button to move onto other pages, and a 'Go Back'␣
↪button to return to the previous page if necessary.")
        label.set_line_wrap(True)
        box.pack_start(label, True, True, 0)
        self.set_page_complete(box, True)

        self.complete = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        self.append_page(self.complete)
        self.set_page_type(self.complete, Gtk.AssistantPageType.PROGRESS)
        self.set_page_title(self.complete, "Page 3: Progress")
        label = Gtk.Label(label="A 'Progress' page is used to prevent changing pages␣
↪within the Assistant before a long-running process has completed. The 'Continue'␣
↪button will be marked as insensitive until the process has finished. Once finished,␣
↪the button will become sensitive.")
        label.set_line_wrap(True)
        self.complete.pack_start(label, True, True, 0)
        checkbutton = Gtk.CheckButton(label="Mark page as complete")
        checkbutton.connect("toggled", self.on_complete_toggled)
        self.complete.pack_start(checkbutton, False, False, 0)

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        self.append_page(box)
        self.set_page_type(box, Gtk.AssistantPageType.CONFIRM)
```

(continues on next page)

(continued from previous page)

```
        self.set_page_title(box, "Page 4: Confirm")
        label = Gtk.Label(label="The 'Confirm' page may be set as the final page in␣
↪the Assistant, however this depends on what the Assistant does. This page provides␣
↪an 'Apply' button to explicitly set the changes, or a 'Go Back' button to correct␣
↪any mistakes.")
        label.set_line_wrap(True)
        box.pack_start(label, True, True, 0)
        self.set_page_complete(box, True)

        box = Gtk.Box(orientation=Gtk.Orientation.VERTICAL)
        self.append_page(box)
        self.set_page_type(box, Gtk.AssistantPageType.SUMMARY)
        self.set_page_title(box, "Page 5: Summary")
        label = Gtk.Label(label="A 'Summary' should be set as the final page of the␣
↪Assistant if used however this depends on the purpose of your Assistant. It␣
↪provides information on the changes that have been made during the configuration or␣
↪details of what the user should do next. On this page only a Close button is␣
↪displayed. Once at the Summary page, the user cannot return to any other page.")
        label.set_line_wrap(True)
        box.pack_start(label, True, True, 0)
        self.set_page_complete(box, True)

    def on_apply_clicked(self, *args):
        print("The 'Apply' button has been clicked")

    def on_close_clicked(self, *args):
        print("The 'Close' button has been clicked")
        Gtk.main_quit()

    def on_cancel_clicked(self, *args):
        print("The Assistant has been cancelled.")
        Gtk.main_quit()

    def on_complete_toggled(self, checkbutton):
        assistant.set_page_complete(self.complete, checkbutton.get_active())

assistant = Assistant()
assistant.show_all()

Gtk.main()
```

Download: `Assistant`

DRAWINGAREA

A DrawingArea is used to create custom interface elements which can be displayed and interacted with by the user. It provides a blank canvas which can be drawn on.

130.1 Constructor

The DrawingArea can be constructed using the following:

```
drawingarea = Gtk.DrawingArea()
```

130.2 Methods

The size of the DrawingArea can be set with:

```
drawingarea.set_size_request(width, height)
```

A *width* and *height* value can be specified to indicate the number of pixels in size the widget should take.

CHAPTER

ONE

PLUG

A Plug object allows an interface to be embedded in a parent (known as a *Socket*) to provide a frontend which is run in a separate process.

Note: A Plug and Socket can also be used on an X11 supported system (e.g. Linux, BSD, Solaris).

131.1 Constructor

The Plug can be constructed using the following:

```
plug = Gtk.Plug.new(socket_id)
```

When a Socket is created, it will return a *socket_id* value which identifies itself uniquely. The *socket_id* should be converted into an integer value.

131.2 Methods

The ID number of a Plug can be retrieved via:

```
id = plug.get_id()
```

To check whether a Plug is currently embedded in a Socket use:

```
embedded = plug.get_embedded()
```

When the Plug is embedded, the *embedded* value returns `True`.

131.3 Signals

The commonly used signals of an Plug are:

```
"embedded" (plug)
```

An “embedded” event is emitted from the Plug when it is attached to a Socket. When the signal occurs, the Plug is passed as part of the event.

131.4 Example

Below is an example of a Plug:

```
#!/usr/bin/env python3

from gi.repository import Gtk
import sys

def embed_event(widget):
    print("A plug has been embedded")

if len(sys.argv) == 2:
    socket_id = sys.argv[1]
    socket_id = int(socket_id)

plug = Gtk.Plug.new(socket_id)
plug.connect("embedded", embed_event)
plug.connect("destroy", Gtk.main_quit)

print("Plug ID:", plug.get_id())

entry = Gtk.Entry()
plug.add(entry)

plug.show_all()

Gtk.main()
```

Download: `Plug`

CHAPTER

TWO

SOCKET

A Socket object provides an object for embedding a *Plug*. The Socket is run in a separate process, and the functionality is transparent to the user.

132.1 Constructor

The Socket can be constructed using the following:

```
socket = Gtk.Socket()
```

132.2 Methods

To retrieve the window ID of the Socket call:

```
id = socket.get_id()
```

When `.get_id()` is called, the unique identifier is returned to the *id* variable.

132.3 Signals

The commonly used signals of an Socket are:

```
"plug-added" (socket)
"plug-removed" (socket)
```

The `"plug-added"` and `"plug-removed"` signals are emitted when the Plug is connected or disconnected from the Socket. In both cases, the Socket on which the Plug was connected is passed.

132.4 Example

Below is an example of a Socket:

```
#!/usr/bin/env python3

from gi.repository import Gtk
```

(continues on next page)

(continued from previous page)

```
import sys

def plug_event(widget):
    print("A plug has been inserted")

window = Gtk.Window()
window.set_default_size(200, 200)
window.connect("destroy", Gtk.main_quit)

socket = Gtk.Socket()
socket.connect("plug-added", plug_event)
window.add(socket)

print("Socket ID:", socket.get_id())

if len(sys.argv) == 2:
    pid = int(sys.argv[1])
    socket.add_id(pid)

window.show_all()

Gtk.main()
```

Download: `Socket`

CHAPTER

THREE

WINDOWGROUP

The WindowGroup class allows *Window* objects to behave like separate applications. When added to a group, of which a Window can be a member of one group at any one time, the effect of grabs are restricted.

133.1 Constructor

The WindowGroup object is constructed using:

```
windowgroup = Gtk.WindowGroup()
```

133.2 Methods

Additions to the group can be made via:

```
windowgroup.add_window(window)
```

A Window object can be removed from the group with:

```
windowgroup.remove_window(window)
```

To list all the Window objects currently associated with a group, call:

```
windowgroup.list_windows()
```

CHAPTER

FOUR

APPLICATION

An Application class handles various common functions for creating a program, such as integration with the desktop, session management and initialisation.

134.1 Constructor

The Application can be constructed using the following:

```
application = Gtk.Application()
```

134.2 Methods

A *Window* can be added to and removed from an Application via:

```
application.add_window(window)
application.remove_window(window)
```

To retrieve a list of Window's attached to an application call:

```
application.get_windows()
```

The active window can be found using:

```
application.get_active_window()
```

A menubar can be defined for the Application after it has been initialised:

```
application.set_menubar(menubar)
```

The *menubar* object needs to be a MenuModel object.

Another type of menu is the App Menu, which contains some common functions such as Quit or Preferences. This can be added using:

```
application.set_app_menu(menu)
```

Again, the *menu* object specified should be a MenuModel.

134.3 Signals

The commonly used signals of an Application are:

```
"window-added" (application, window)
"window-removed" (application, window)
```

The `"window-added"` and `"window-removed"` signals emit from the Application when the `.add_window()` and `.remove_window()` methods are called.

134.4 Example

Below is an example of a Application:

```
#!/usr/bin/env python3

from gi.repository import Gtk
from gi.repository import Gio
import sys

class ApplicationWindow(Gtk.ApplicationWindow):
    def __init__(self, application):
        Gtk.Window.__init__(self, application=application)
        self.set_title("Application")
        self.set_default_size(200, 200)

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

        menubutton = Gtk.MenuButton()
        grid.attach(menubutton, 0, 0, 1, 1)

        menumodel = Gio.Menu()
        menubutton.set_menu_model(menumodel)
        menumodel.append("New", "app.new")
        menumodel.append("Quit", "app.quit")

class Application(Gtk.Application):
    def __init__(self):
        Gtk.Application.__init__(self)

    def do_activate(self):
        window = ApplicationWindow(self)
        window.show_all()

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

        new_action = Gio.SimpleAction.new("new", None)
        new_action.connect("activate", self.new_callback)
        self.add_action(new_action)

        quit_action = Gio.SimpleAction.new("quit", None)
        quit_action.connect("activate", self.quit_callback)
        self.add_action(quit_action)
```

(continues on next page)

(continued from previous page)

```
    def new_callback(self, action, parameter):
        print("You clicked New")

    def quit_callback(self, action, parameter):
        print("You clicked Quit")
        self.quit()

application = Application()
exit_status = application.run(sys.argv)
sys.exit(exit_status)
```

Download: `Application`

CHAPTER

FIVE

CLIPBOARD

The clipboard provides a temporary memory store of data, and is typically used to allow objects such as text, images, audio, video or files to be transferred between locations or applications. The most typical way of using the clipboard involves cutting/copying and subsequently pasting in the desired location, and this is done using keyboard shortcuts or a widget such as a *Button* or *MenuItem*.

On the Linux system, there are two clipboards. The most commonly used is the standard clipboard used to store images, files, etc. in addition to text. The other is the primary which holds the currently highlighted text and is accessed by middle-clicking in the desired location.

Note: A clipboard and its associated methods are quite complex and will require good understanding of GTK+.

135.1 Constructor

The Clipboard can be constructed using the following:

```
clipboard = Gtk.Clipboard.get(selection)
```

The selection value should be set to either `Gdk.SELECTION_CLIPBOARD` or `Gdk.SELECTION_PRIMARY`. The `Gdk.SELECTION_CLIPBOARD` item allows content to be transferred to and from the general clipboard. Alternatively, `Gdk.SELECTION_PRIMARY` is used to capture content when it is highlighted.

Note: The use `Gdk.SELECTION_CLIPBOARD` or `Gdk.SELECTION_PRIMARY`, you will need to import the Gdk module before calling a Gdk constant.

135.2 Methods

To clear the current contents of the Clipboard use:

```
clipboard.clear()
```

Note: The .clear() method will clear the contents of the clipboard regardless of which application stored the data.

Setting text on the Clipboard can be achieved via:

```
clipboard.set_text(text, length)
```

The *text* value is the string of text which is to be stored on the clipboard. The *length* value allows for the number of characters to be limited. If all the content should be stored, use `-1`.

Images can also be stored using:

```
clipboard.set_image(image)
```

The *image* paramter should be a Pixbuf object representing the image.

By default, when an application closes, any data saved to the clipboard using it will be cleared. To store the data after the application closes call:

```
clipboard.store()
```

135.3 Example

Below is an example of a Clipboard:

```
#!/usr/bin/env python3

from gi.repository import Gtk, Gdk

class Clipboard(Gtk.Window):
    def __init__(self):
        Gtk.Window.__init__(self)
        self.connect("destroy", Gtk.main_quit)

        self.clipboard = Gtk.Clipboard.get(Gdk.SELECTION_CLIPBOARD)

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

        self.entry = Gtk.Entry()
        grid.attach(self.entry, 0, 0, 1, 1)

        buttonCutText = Gtk.Button(label="Cut Text")
        buttonCutText.connect("clicked", self.on_copy_text, "cut")
        grid.attach(buttonCutText, 1, 0, 1, 1)

        buttonCopyText = Gtk.Button(label="Copy Text")
        buttonCopyText.connect("clicked", self.on_copy_text, "copy")
        grid.attach(buttonCopyText, 2, 0, 1, 1)

        buttonPasteText = Gtk.Button(label="Paste Text")
        buttonPasteText.connect("clicked", self.on_paste_text)
        grid.attach(buttonPasteText, 3, 0, 1, 1)

    def on_copy_text(self, button, action):
        content = self.entry.get_text()

        if action == "cut":
            self.entry.set_text("")
```

(continues on next page)

(continued from previous page)

```
        self.clipboard.set_text(content, -1)

    def on_paste_text(self, button):
        content = self.clipboard.wait_for_text()
        self.entry.set_text(content)

window = Clipboard()
window.show_all()

Gtk.main()
```

Download: `Clipboard`

CHAPTER

SIX

PRINTOPERATION

The PrintOperation object is a key part of the printing functionality of GTK+, and connects to each of the other printing functionality such as the dialogs, settings, and page setup details.

136.1 Constructor

The PrintOperation is constructed via:

```
printoperation = Gtk.PrintOperation()
```

136.2 Methods

The job name is settable using:

```
printoperation.set_job_name(name)
```

A job name is useful for the end user to identify the document being printed. If no job name is specified, GTK+ picks a default one based on numbering of successive jobs.

The printing functionality supports exporting the print job to file (e.g. PDF) rather than an actual device. The default filename of the exported document can be set with:

```
printoperation.set_export_filename(filename)
```

The units used by the print job for sizing purposes can be set with:

```
printoperation.set_unit(unit)
```

The *unit* parameter should be set to one of the following:

- `Gtk.Unit.NONE`
- `Gtk.Unit.POINTS`
- `Gtk.Unit.INCH`
- `Gtk.Unit.MM`

The print settings associated with the job can be attached by calling:

```
printoperation.set_print_settings(settings)
```

The *settings* parameter should be set to the *PrintSettings* object for the job, which contains information such as paper size and orientation, whether to use colour, or the number of copies.

To configure whether the progress of the print job is shown, use:

```
printoperation.set_show_progress(show_progress)
```

When *show_progress* is set to `True`, a progress bar will be shown within a dialog.

Displaying status messages on the state of the printer can be important in some cases. The status can be tracked via:

```
printoperation.set_track_print_status(track_status)
```

When *track_status* is set to `True`, the status of the job queue may be reported back. For instance, this could be used to display "Out of paper".

CHAPTER

SEVEN

PRINTER

The Printer object represents a physical or virtual printer. Dealing with it directly is only required if the non-portable *PrintUnixDialog* is being used.

Use of the object allows information to be retrieved such as location, description, number of queued jobs, etc. It is also used in conjunction with *PrintJob* to create a print job.

137.1 Methods

To return the assigned name of the printer call:

```
printer.get_name()
```

Other useful information can be obtained with:

```
printer.get_location()
printer.get_description()
```

To obtain the status message for the printer use:

```
printer.get_state_message()
```

The number of queued jobs can be obtained from the device via:

```
printer.get_job_count()
```

Checking whether the printer is currently active or paused is done by:

```
printer.is_active()
printer.is_paused()
```

A method is also available to check whether the printer is currently accepting jobs by:

```
printer.is_accepting_jobs()
```

To check whether the printer is set as the default device for the user call:

```
printer.is_default()
```

Functions are also available to check whether the printer accepts PDF or PS formats using:

```
printer.accepts_pdf()
printer.accepts_ps()
```

Both *methods* will return `True` if they accept the format.

A list of paper types which the printer accepts can be obtained using the method:

```
printer.list_papers()
```

If the paper types supported by the printer are not available, the returned list will be empty.

CHAPTER

EIGHT

PRINTUNIXDIALOG

The PrintUnixDialog implements a dialog for platforms which do not provide a native dialog. It provides functionality for setting up the print options such as printer to output the document to, the number of copies, the page orientation, and image quality settings.

138.1 Constructor

A PrintUnixDialog is able to be constructed using:

```
printunixdialog = Gtk.PrintUnixDialog()
```

138.2 Methods

Retrieval of the selected printer object can be made using the method:

```
printunixdialog.get_selected_printer()
```

The `.get_selected_printer()` method returns a *Printer* object.

In some cases, the user will be able to select only a portion of the document to print. The dialog can be told that it should support selections, and provide the option using:

```
printunixdialog.set_support_selection(support_selection)
```

The capabilities of the print dialog can be customised with:

```
printunixdialog.set_manual_capabilities(capabilities)
```

A large number of *capabilities* are offered which change the offered options:

- `Gtk.PrintCapability.PAGE_SET` - offer printing of odd/even pages.
- `Gtk.PrintCapability.COPIES` - allow setting of number of copies.
- `Gtk.PrintCapability.COLLATE` - provide collation of multiple copies.
- `Gtk.PrintCapability.REVERSE` - set whether outputted job is done in reverse.
- `Gtk.PrintCapability.SCALE` - scale the output.
- `Gtk.PrintCapability.GENERATE_PDF` - allow sending of document to PDF output.
- `Gtk.PrintCapability.GENERATE_PS` - allow sending of document to PS output.

- `Gtk.PrintCapability.PREVIEW` - provide option to preview output.
- `Gtk.PrintCapability.NUMBER_UP` - setting to print multiple sheets onto a single page.
- `Gtk.PrintCapability.NUMBER_UP_LAYOUT` - setting to allow rearrange multiple sheets onto a single page.

A custom tab can be added to the printing dialog which allows placement of additional widgets:

```
printunixdialog.add_custom_tab(child, label)
```

The *child* parameter indicates the widget to be displayed in the tab, with the *label* parameter identifying the tabs purpose.

CHAPTER

NINE

PRINTSETTINGS

The PrintSettings object stores the configuration for the print job. Example settings include use of duplex, output bin, and number of copies amongst others.

139.1 Constructor

Construction of the PrintSettings object is done via:

```
printsettings = Gtk.PrintSettings()
```

139.2 Methods

The Printer object associated with the PrintSettings can be retrieved with:

```
printsettings.get_printer()
```

A Printer object can also be defined:

```
printsettings.set_printer(printer)
```

A range of print settings can be defined with:

```
printsettings.set_n_copies(copies)
printsettings.set_reverse(reverse)
printsettings.set_collate(collate)
printsettings.set_use_color(color)
```

Print quality options can be defined via the method:

```
printsettings.set_quality(quality)
```

The *quality* value should be set to one of the following constants:

- `Gtk.PrintQuality.LOW`
- `Gtk.PrintQuality.NORMAL`
- `Gtk.PrintQuality.HIGH`
- `Gtk.PrintQuality.DRAFT`

The current quality setting can also be retrieved if required:

```
printsettings.get_quality()
```

The paper size in use by the printer can be defined with:

```
printsettings.set_paper_size(paper_size)
```

The *paper_size* constant should be set to an appropriate *PaperSize* object.

Custom paper sizes can also be entered via:

```
printsettings.set_paper_width(width, unit)
printsettings.set_paper_height(height, unit)
```

The *width* and *height* arguments should be set to a decimal value. The *unit* determines the measurement and should be set to a constant of:

- `Gtk.Unit.NONE`
- `Gtk.Unit.POINTS`
- `Gtk.Unit.INCH`
- `Gtk.Unit.MM`

CHAPTER

PRINTJOB

The PrintJob object represents a print job sent to the printer.

140.1 Constructor

The constructor for the PrintJob is:

```
printjob = Gtk.PrintJob(title, printer, printsettings, pagesetup)
```

140.2 Methods

The title of the PrintJob can be retrieved with:

```
printjob.get_title()
```

The number of pages to be printed is settable using:

```
printjob.set_pages(pages)
```

A number of features about the print output can be fetched using:

```
printjob.get_reverse()
printjob.get_collate()
printjob.get_rotate()
printjob.get_num_copies()
```

The print settings are also settable via:

```
printjob.set_reverse(reverse)
printjob.set_collate(collate)
printjob.set_rotate(rotate)
printjob.set_num_copies(copies)
```

Retrieval of a print job status is done with the method:

```
printjob.get_status()
```

The *PrintSettings* or *Printer* objects can be requested using:

```
printjob.get_settings()
printjob.get_printer()
```

CHAPTER

ONE

PAPERSIZE

The PaperSize object handles paper sizing, and is commonly used with the printing framework of GTK+.

141.1 Constructor

The PaperSize is constructed using:

```
papersize = Gtk.PaperSize(name)
```

The *name* parameter specifies the size of the paper object to be created using the PWG 5101.1-2002 paper name convention. This is parsed by GTK+ to set the values for the object. Common examples include “a4”, “a3”, “na_letter”, etc.

If the appropriate size is not available, custom values can be specified with:

```
papersize = Gtk.PaperSize(name, display_name, width, height, unit)
```

The *name* object is the paper name to be used by this object, while *display_name* is a human-readable name. The *width* and *height* values indicate the size of the paper object created while the *unit* parameter sets the units used by the width and height. This should be set to one of:

- `Gtk.Unit.POINTS`
- `Gtk.Unit.INCH`
- `Gtk.Unit.MM`

141.2 Methods

Setting the size of the paper is done via:

```
papersize.set_size(width, height, unit)
```

The *width* and *height* parameters indicate the size of the paper. The *unit* should be set to one of:

- `Gtk.Unit.POINTS`
- `Gtk.Unit.INCH`
- `Gtk.Unit.MM`

Retrieval of the specified width and height is done by calling:

```
papersize.get_width()
papersize.get_height()
```

The name and display name used by the PaperSize object can be fetched after construction with:

```
papersize.get_name()
papersize.get_display_name()
```

An existing PaperSize can be copied to a new object with:

```
papersize.copy(size)
```

To check if two PaperSize objects are equal, use:

```
papersize.is_equal(size1, size2)
```

CHAPTER

TWO

PAGESETUP

The PageSetup object stores the page size, orientation and margins. These are often retrieved from the page setup dialog which is then passed on to the PrintOperation for printing.

Note: The information relating to margins pertains to the printer margins, which the printer is not able to print within. They do not refer to margins such as those found in a word processor.

142.1 Methods

The orientation of a page can be adjusted via:

```
pagesetup.set_orientation(orientation)
```

The *orientation* should be set to one of the PageOrientation constants:

- `Gtk.PageOrientation.PORTRAIT`
- `Gtk.PageOrientation.LANDSCAPE`
- `Gtk.PageOrientation.REVERSE_PORTRAIT`
- `Gtk.PageOrientation.REVERSE_LANDSCAPE`

Alternatively, the page orientation can be retrieved with:

```
pagesetup.get_orientation()
```

The paper size is settable via a *PaperSize* object using:

```
pagesetup.set_paper_size(papersize)
```

Retrieval of the

The margin values can be set on the page with:

```
pagesetup.set_top_margin(margin, unit)
pagesetup.set_bottom_margin(margin, unit)
pagesetup.set_left_margin(margin, unit)
pagesetup.set_right_margin(margin, unit)
```

The *margin* value of each method should be set to the number of pixels reserved for the margin. A *unit* value should also be specified from:

- `Gtk.Units.POINTS`

- `Gtk.Units.INCH`
- `Gtk.Units.MM`

The margin sizes can also be fetched via:

```
pagesetup.get_top_margin(unit)
pagesetup.get_bottom_margin(unit)
pagesetup.get_left_margin(unit)
pagesetup.get_right_margin(unit)
```

As with setting the margin size, a *unit* should also be specified when getting the sizes.

The width and height of the paper can be retrieved via:

```
pagesetup.get_paper_width(unit)
pagesetup.get_paper_height(unit)
```

The page width and height can also be retrieved using the methods:

```
pagesetup.get_page_width(unit)
pagesetup.get_page_height(unit)
```

CHAPTER

THREE

DRAG AND DROP

Drag and drop is a complex topic and requires careful setup to allow transferring of information to ensure the drag and drop operations work correctly. The drag and drop operation consists of a drag source and a drop destination, typically both consisting of a widget such as a Button or TreeView.

Note: The drag and drop functionality of GTK+ requires a good understanding of the toolkit.

143.1 Basic Functions

Basic drag and drop functionality is available, and can be used between simpler widgets such as a Label.

143.2 Advanced Functions

143.3 Signals

A number of signals can be setup on the source and destination objects to control the functionality.

143.3.1 Source

The source is the object from where the drag and drop operation is started.

- `"drag-begin"` - commonly used to setup an icon which will be displayed to the user when the drag operation is started.
- `"drag-data-get"` - grab the data which will be transferred to the destination.
- `"drag-data-delete"` - delete the data if the drag and drop operation will move rather than copy.
- `"drag-end"` - undo any setup begun with the `"drag-begin"` signal.

143.3.2 Destination

A destination is an object which is the receiver of the drag and drop operation.

CHAPTER

FOUR

PAGESETUPUNIXDIALOG

The PageSetupUnixDialog provides a dialog for platforms which don't provide a native alternative, and offers options on setting up pages for printing. Its features allow for modifying the paper size or the page output orientation.

144.1 Constructor

The construction of the PageSetupUnixDialog dialog is done with:

```
pagesetupunixdialog = Gtk.PageSetupUnixDialog()
```

144.2 Methods

The *PageSetup* object can be attached to the dialog via:

```
pagesetupunixdialog.set_page_setup(pagesetup)
```

The PageSetup can also be retrieved using:

```
pagesetupunixdialog.get_page_setup()
```

A *PrintSettings* object can also be specified with:

```
pagesetupunixdialog.set_print_settings(printsettings)
```

Retrieval of the PrintSettings can also be fetched via:

```
pagesetupunixdialog.get_print_settings()
```

CHAPTER

FIVE

BUILDER

Building applications manually can be difficult and time-consuming. An alternative method is to use a program such as Glade to build the interface using a graphical application in a "What You See Is What You Get" manner. The resulting file contains all the widgets and associated properties which are then automatically built at runtime.

The use of a WYSIWYG editor is beneficial as it requires less code to be written, interface changes are able to be made quicker, and people with no programming skills can still produce interfaces.

145.1 Constructor

The Builder object is constructed using:

```
builder = Gtk.Builder()
```

145.2 Methods

Primarily, the interface will be added from the Glade-produced file:

```
builder.add_from_file(filename)
```

Alternatively, it can be added from a string with:

```
builder.add_from_string(string)
```

An individual object can be obtained via:

```
builder.get_object(object)
```

The *object* string should point to a valid item name held within the Glade file.

All the items held by the Builder can be fetched and placed into a list with:

```
builder.get_objects()
```

In combination with a dictionary, the signals can be connected automatically using the method:

```
builder.connect_signals(handlers)
```

CHAPTER

SIX

COMMON METHODS

There are a number of methods which apply to many widgets. These include:

```
.set_sensitive(sensitive)
```

Setting *sensitive* to `False` greys-out a widget and prevents the user from using it.

```
.set_visible(visible)
```

The *visible* parameter when set to `False` removes the widget from view from the user.

```
.get_visible()
```

The `.get_visible()` method returns `True` when the widget is being shown, and `False` when hidden.

```
.show()
.show_all()
.hide()
```

The `.show()` and `.show_all()` methods display widgets on screen, however `.show_all()` will display the parent and all subsequent child widgets. The `.hide()` method prompts GTK+ to hide the widget from display.

```
.destroy()
```

Calling `.destroy()` deletes the widget and frees up the resources it was using.

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

Using `.set_size_request()` allows configuring a widget based on the width and height in pixels.

```
.is_focus()
```

To check whether a widget has the focus, call `.is_focus()`. If the widget is currently the focus, `True` is returned.

```
.set_can_focus(can_focus)
```

The *can_focus* parameter when set to `False` prevents the widget from accepting the input focus if required.

```
.set_vexpand(expand)
.set_hexpand(expand)
```

Items added to a *Grid* by default are shrunk to the size of the content they contain. When the *expand* parameter is set to `True`, the item is sized to fit the space vertically, horizontally or both.

CHAPTER

SEVEN

GLOSSARY

Below are some of the common terms used throughout this tutorial.

Accelerator a key-combination which provides access to functions via the keyboard

Action See "Event"

Argument See "Parameter"

Boolean A true or false value

Child A widget which relates to a parent (e.g. a Label is a child of a HBox)

Container Object which allows other widgets to be contained within

Constant A fixed value which provides a default set of configuration details

Constructor Piece of code which creates a widget for use in your application and gives it a name

Event Something which occurs within the application, whether caused by the user or is done programatically

Homogeneous All equal or considered the same

Insensitive The official term for "greyed-out" or disabled, preventing the user from interacting with it

Interface An object, which is not visible to the end-user but is used to provide common functions used by the visible widgets

Method A piece of code which allows an object to be customised or configured

Mnemonic An accessibility feature which allows access to functions via the keyboard. These are identified by an underscore under a character

Object An item which is constructed but is not visible to the user

Packing The act of placing widgets within others to create an interface

Parameter A value which is provided to a constructor or method to indicate a setting or configuration

Parent A widget which contains a child (e.g. a Window is the parent to a VBox)

Pixbuf The internal image format used by GTK+. Graphics used within GTK+ are usually converted to Pixbuf format

Property Similar to a method in that it controls functionality of a widget or object, however usally allows for fined-grain control

Signal Emitted when an action or event takes place

Signature The values emitted by the signal when an event has occured

Widget An item which the user interacts with on screen

INDEX