PyGTK tutorial

From: http://www.zetcode.com/tutorials/pygtktutorial/

This is PyGTK tutorial. The PyGTK tutorial is suitable for beginner and more advanced programmers.

Introduction to PyGTK

In this part of the PyGTK programming tutorial, we will talk about the PyGTK GUI library and Python programming language in general.

About this tutorial

This is PyGTK programming tutorial. It has been created and tested on Linux. The PyGTK programming tutorial is suited for novice and more advanced programmers.

PyGTK

PyGTK is a set of Python wrappers for the GTK+ GUI library. It offers a comprehensive set of graphical elements and other useful programming facilities for creating desktop applications. It is a part of the GNOME project. PyGTK is free software and licensed under the LGPL. Original autor of PyGTK is **James Henstridge**. PyGTK is very easy to use, it is ideal for rapid prototyping. Currently, PyGTK is one of the most popular bindings for the GTK+ library.

PyGTK consists of several modules.

GObject is a base class providing the common attributes and functions for PyGTK classes. ATK is the accessibility toolkit. This toolkit provides tools which help physically challenged people work with computers. GTK is the user interface module. The Pango is a library which is used to work with text and internationalization. Cairo is a library for creating 2D vector graphics. Glade is used to build GUI interfaces from XML descriptions.

Python

Python is a dynamic object-oriented programming language. It is a general purpose programming language. It can be used for many kinds of software development. The design purpose of the Python language emphasizes programmer productivity and code readability. Python was initially developed by Guido van Rossum. It was first released in 1991. Python was inspired by ABC, Haskell, Java, Lisp, Icon and Perl programming languages. Python is a high level, general purpose, multiplatform, interpreted language. Python is a minimalistic language. One of it's most visible features is that it does not use semicolons nor brackets. Python uses indentation instead. There are two main branches of Python currently. Python 2.x and Python 3.x. Python 3.x breaks backward compatibility with previous releases of Python. It was created to correct some design flaws of the language and make the language more clean. The most recent version of Python 2.x is 2.7.1, and of Python 3.x 3.1.3. This tutorial is writtein in Python 2.x. Today, Python is maintained by a large group of volunteers worldwide.

GTK+

The GTK+ is a library for creating graphical user interfaces. The library is created in C programming language. The GTK+ library is also called the GIMP Toolkit. Originally, the library was created while

developing the GIMP image manipulation program. Since then, the GTK+ became one of the most popular toolkits under Linux and BSD Unix. Today, most of the GUI software in the open source world is created in Qt or in GTK+. The GTK+ is an object oriented application programming interface. The object oriented system is created with the Glib object system, which is a base for the GTK+ library. The GObject also enables to create language bindings for various other programming languages. Language bindings exist for C++, Python, Perl, Java, C# and other programming languages.

Gnome and XFce desktop environments have been created using the GTK+ library. SWT and wxWidgets are well known programming frameworks, that use GTK+. Prominent software applications that use GTK+ include Firefox or Inkscape.

Sources

- pygtk.org
- wikipedia.org

First steps in PyGTK

In this part of the PyGTK programming tutorial, we will do our first steps in programming. We will create simple programs.

Simple example

The first code example is a very simple one.

center.py

```
#!/usr/bin/python
# ZetCode PyGTK tutorial
#
# This is a trivial PyGTK example
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009
import gtk
class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()
```

```
        self.connect("destroy", gtk.main_quit)
        self.set_size_request(250, 150)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show()
PyApp()
gtk.main()
```

This code shows a centered window.

```
import gtk
```

We import the gtk module. Here we have objects to create GUI applications.

```
class PyApp(gtk.Window):
```

Our application is based on the **PyApp** class. It inherits from the **Window**.

```
def __init__(self):
    super(PyApp, self).__init__()
```

This is the constructor. It builds our application. It also calls it's parent constructor through the **super()** call.

```
self.connect("destroy", gtk.main_quit)
```

We connect the **destroy** signal to the **main_quit()** function. The **destroy** signal is called when we click on the close button in the titlebar or press Alt + F4. The window is being destroyed, but the application is not. You can see it, if you launch the example from the command line. By calling the **main_quit()** we quit the application for good.

```
self.set_size_request(250, 150)
```

We set the size of the window to 250x150px.

```
self.set_position(gtk.WIN_POS_CENTER)
```

This line centers the window on the screen.

```
self.show()
```

Now we show the window. The window is not visible, until we call the **show()** method.

```
PyApp()
gtk.main()
```

We create the instance of our program and start the main loop.

Icon

In the next example, we show the application icon. Most window managers display the icon in the left corner of the titlebar and also on the taskbar.

icon.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows an icon
# in the titlebar of the window
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk, sys

class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Icon")
        self.set_size_request(250, 150)
        self.set_position(gtk.WIN_POS_CENTER)

        try:
            self.set_icon_from_file("web.png")
        except Exception, e:
            print e.message
            sys.exit(1)

        self.connect("destroy", gtk.main_quit)

        self.show()

PyApp()
gtk.main()
```

The code example shows the application icon.

```
 self.set_title("Icon")
```

We set a title for the window.

```
 self.set_icon_from_file("web.png")
```

The `set_icon_from_file()` method sets an icon for the window. The image is loaded from disk in the current working directory.

Figure: Icon

Buttons

In the next example, we will further enhance our programming skills with the PyGTK library.

buttons.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows four buttons
# in various modes
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk

class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Buttons")
        self.set_size_request(250, 200)
        self.set_position(gtk.WIN_POS_CENTER)

        btn1 = gtk.Button("Button")
        btn1.set_sensitive(False)
        btn2 = gtk.Button("Button")
        btn3 = gtk.Button(stock=gtk.STOCK_CLOSE)
        btn4 = gtk.Button("Button")
        btn4.set_size_request(80, 40)

        fixed = gtk.Fixed()
```

```
        fixed.put(btn1, 20, 30)
        fixed.put(btn2, 100, 30)
        fixed.put(btn3, 20, 80)
        fixed.put(btn4, 100, 80)

        self.connect("destroy", gtk.main_quit)

        self.add(fixed)
        self.show_all()


PyApp()
gtk.main()
```

We show four different buttons on the window. We will see a difference between container widgets and child widgets and will change some properties of child widgets.

```
btn1 = gtk.Button("Button")
```

A **Button** is a child widget. Child widgets are placed inside containers.

```
btn1.set_sensitive(False)
```

We make this button insensitive. This means, we cannot click on it. Nor it can be selected, focused etc. Graphically the widget is grayed out.

```
btn3 = gtk.Button(stock=gtk.STOCK_CLOSE)
```

The third button shows an image inside it's area. The PyGTK library has a built-in stock of images, that we can use.

```
btn4.set_size_request(80, 40)
```

Here we change the size of the button.

```
fixed = gtk.Fixed()
```

Fixed widget is a non visible container widget. It's purpose is to contain other child widgets.

```
fixed.put(btn1, 20, 30)
fixed.put(btn2, 100, 30)
...
```

Here we place button widgets inside fixed container widget.

```
self.add(fixed)
```

We set the **Fixed** container to be the main container for our **Window** widget.

```
self.show_all()
```

We can either call show_all() method, or we call show() method on each of the widgets. Including containers.

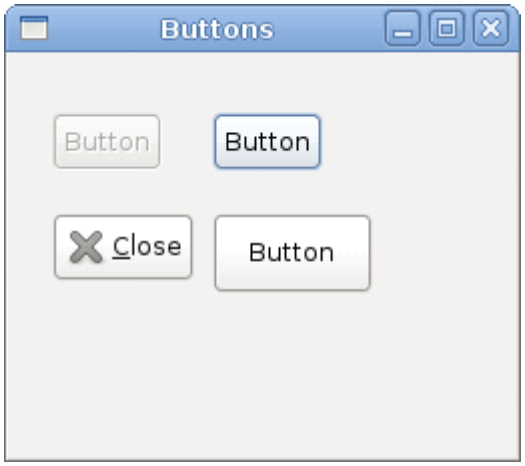


Figure: Buttons

Tooltip

A tooltip is a hint on a widget in the applications. Can be used to provide additional help.

tooltips.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This code shows a tooltip on
# a window and a button
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Tooltips")
        self.set_size_request(250, 200)
        self.set_position(gtk.WIN_POS_CENTER)
```

```
        self.connect("destroy", gtk.main_quit)

        self.fixed = gtk.Fixed()
        self.add(self.fixed)

        button = gtk.Button("Button")
        button.set_size_request(80, 35)

        self.fixed.put(button, 50, 50)

        self.set_tooltip_text("Window widget")
        button.set_tooltip_text("Button widget")

        self.show_all()

PyApp()
gtk.main()
```

In this example we set a tooltip for a window and for a button.

```
self.set_tooltip_text("Window widget")
button.set_tooltip_text("Button widget")
```

The `set_tooltip_text()` does the job.

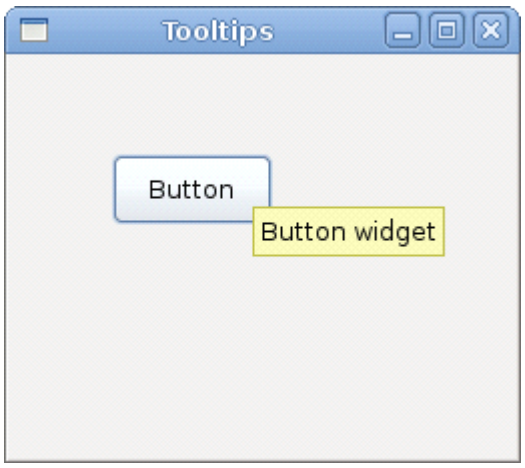


Figure: Tooltips

In this chapter, we created first programs in PyGTK programming library.

Layout management in PyGTK

In this chapter of the PyGTK tutorial, we will show how to lay out our widgets in windows or dialogs.

When we design the GUI of our application, we decide what widgets we will use and how we will organize those widgets in the application. To organize our widgets, we use specialized non visible widgets called **layout containers**. In this chapter, we will mention **Alignment**, **Fixed**, **VBox** and **Table**.

Fixed

The **Fixed** container places child widgets at fixed positions and with fixed sizes. This container performs no automatic layout management. In most applications, we don't use this container. There are some specialized areas, where we use it. For example games, specialized applications that work with diagrams, resizable components that can be moved (like a chart in a spreadsheet application), small educational examples.

fixed.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates a Fixed
# container widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import sys


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Fixed")
        self.set_size_request(300, 280)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        try:
            self.bardejov = gtk.gdk.pixbuf_new_from_file("bardejov.jpg")
            self.rotunda = gtk.gdk.pixbuf_new_from_file("rotunda.jpg")
            self.mincol = gtk.gdk.pixbuf_new_from_file("mincol.jpg")
        except Exception, e:
            print e.message
```

```
            sys.exit(1)


        image1 = gtk.Image()
        image2 = gtk.Image()
        image3 = gtk.Image()

        image1.set_from_pixbuf(self.bardejov)
        image2.set_from_pixbuf(self.rotunda)
        image3.set_from_pixbuf(self.mincol)

        fix = gtk.Fixed()

        fix.put(image1, 20, 20)
        fix.put(image2, 40, 160)
        fix.put(image3, 170, 50)

        self.add(fix)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

In our example, we show three small images on the window. We explicitely specify the x, y coordinates, where we place these images.

```
self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
```

For better visual experience, we change the background color to dark gray.

```
self.bardejov = gtk.gdk.pixbuf_new_from_file("bardejov.jpg")
```

We load the image from a file on the disk.

```
image1 = gtk.Image()
image2 = gtk.Image()
image3 = gtk.Image()

image1.set_from_pixbuf(self.bardejov)
image2.set_from_pixbuf(self.rotunda)
image3.set_from_pixbuf(self.mincol)
```

The **Image** is a widget, that is used to display images. It takes a **Pixbuf** object in the constructor.

```
fix = gtk.Fixed()
```

We create the **Fixed** container.

```
fix.put(image1, 20, 20)
```

We place the first image at x=20, y=20 coordinates.

```
self.add(fix)
```

Finally, we add the **Fixed** container to the Window.

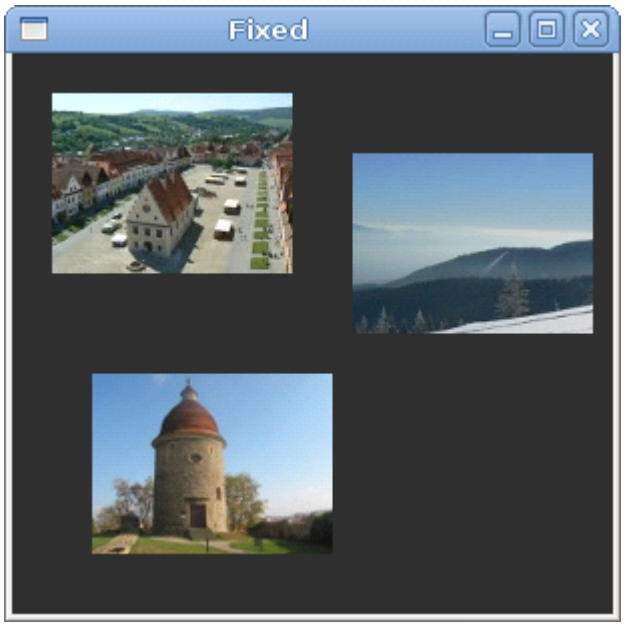


Figure: Fixed

Alignment

The **Alignment** container controls the alignment and the size of it's child widget.

alignment.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to use
# the Alignment widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):
```

```
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Alignment")
        self.set_size_request(260, 150)
        self.set_position(gtk.WIN_POS_CENTER)

        vbox = gtk.VBox(False, 5)
        hbox = gtk.HBox(True, 3)

        valign = gtk.Alignment(0, 1, 0, 0)
        vbox.pack_start(valign)

        ok = gtk.Button("OK")
        ok.set_size_request(70, 30)
        close = gtk.Button("Close")

        hbox.add(ok)
        hbox.add(close)

        halign = gtk.Alignment(1, 0, 0, 0)
        halign.add(hbox)

        vbox.pack_start(halign, False, False, 3)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

In the code example, we place two buttons into the right bottom corner of the window. To accomplish this, we use one horizontal box and one vertical box and two alignment containers.

```
valign = gtk.Alignment(0, 1, 0, 0)
```

This will put the child widget to the bottom.

```
vbox.pack_start(valign)
```

Here we place the Alignment widget into the vertical box.

```
hbox = gtk.HBox(True, 3)
...
ok = gtk.Button("OK")
ok.set_size_request(70, 30)
close = gtk.Button("Close")

hbox.add(ok)
hbox.add(close)
```

We create a horizontal box and put two buttons inside it.

```
halign = gtk.Alignment(1, 0, 0, 0)
halign.add(hbox)

vbox.pack_start(halign, False, False, 3)
```

This will create an alignment container that will place it's child widget to the right. We add the horizontal box into the alignment container and pack the alignment container into the vertical box. We must keep in mind that the alignment container takes only one child widget. That's why we must use boxes.

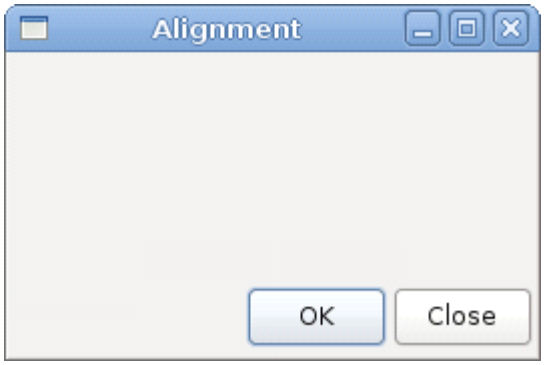


Figure: Alignment

Table

The Table widget arranges widgets in rows and columns.

calculator.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to use
# the Table container widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):

    def __init__(self):
```

```
        super(PyApp, self).__init__()

        self.set_title("Calculator")
        self.set_size_request(250, 230)
        self.set_position(gtk.WIN_POS_CENTER)

        vbox = gtk.VBox(False, 2)

        mb = gtk.MenuBar()
        filemenu = gtk.Menu()
        filem = gtk.MenuItem("File")
        filem.set_submenu(filemenu)
        mb.append(filem)

        vbox.pack_start(mb, False, False, 0)

        table = gtk.Table(5, 4, True)

        table.attach(gtk.Button("Cls"), 0, 1, 0, 1)
        table.attach(gtk.Button("Bck"), 1, 2, 0, 1)
        table.attach(gtk.Label(), 2, 3, 0, 1)
        table.attach(gtk.Button("Close"), 3, 4, 0, 1)

        table.attach(gtk.Button("7"), 0, 1, 1, 2)
        table.attach(gtk.Button("8"), 1, 2, 1, 2)
        table.attach(gtk.Button("9"), 2, 3, 1, 2)
        table.attach(gtk.Button("/"), 3, 4, 1, 2)

        table.attach(gtk.Button("4"), 0, 1, 2, 3)
        table.attach(gtk.Button("5"), 1, 2, 2, 3)
        table.attach(gtk.Button("6"), 2, 3, 2, 3)
        table.attach(gtk.Button("*"), 3, 4, 2, 3)

        table.attach(gtk.Button("1"), 0, 1, 3, 4)
        table.attach(gtk.Button("2"), 1, 2, 3, 4)
        table.attach(gtk.Button("3"), 2, 3, 3, 4)
        table.attach(gtk.Button("-"), 3, 4, 3, 4)

        table.attach(gtk.Button("0"), 0, 1, 4, 5)
        table.attach(gtk.Button("."), 1, 2, 4, 5)
        table.attach(gtk.Button("="), 2, 3, 4, 5)
        table.attach(gtk.Button("+"), 3, 4, 4, 5)

        vbox.pack_start(gtk.Entry(), False, False, 0)
        vbox.pack_end(table, True, True, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

We use the Table widget to create a calculator skeleton.

```
table = gtk.Table(5, 4, True)
```

We create a table widget with 5 rows and 4 columns. The third parameter is the homogenous parameter. If set to true, all the widgets in the table are of same size. The size of all widgets is equal to the largest widget in the table container.

```
table.attach(gtk.Button("Cls"), 0, 1, 0, 1)
```

We attach a button to the table container. To the top-left cell of the table. The first two parameters are the left and right sides of the cell, the last two parameters are the top and bottom sides of the cell.

```
vbox.pack_end(table, True, True, 0)
```

We pack the table widget into the vertical box.

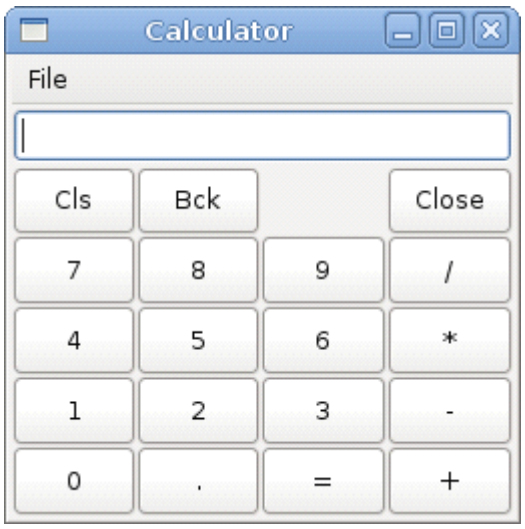


Figure: Calculator skeleton

Windows

Next we will create a more advanced example. We show a window, that can be found in the JDeveloper IDE.

windows.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This is a more complicated layout
# example
#
```

```
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import sys


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Windows")
        self.set_size_request(300, 250)
        self.set_border_width(8)
        self.set_position(gtk.WIN_POS_CENTER)

        table = gtk.Table(8, 4, False)
        table.set_col_spacings(3)

        title = gtk.Label("Windows")

        halign = gtk.Alignment(0, 0, 0, 0)
        halign.add(title)

        table.attach(halign, 0, 1, 0, 1, gtk.FILL,
            gtk.FILL, 0, 0);

        wins = gtk.TextView()
        wins.set_editable(False)
        wins.modify_fg(gtk.STATE_NORMAL, gtk.gdk.Color(5140, 5140, 5140))
        wins.set_cursor_visible(False)
        table.attach(wins, 0, 2, 1, 3, gtk.FILL | gtk.EXPAND,
            gtk.FILL | gtk.EXPAND, 1, 1)

        activate = gtk.Button("Activate")
        activate.set_size_request(50, 30)
        table.attach(activate, 3, 4, 1, 2, gtk.FILL,
            gtk.SHRINK, 1, 1)

        valign = gtk.Alignment(0, 0, 0, 0)
        close = gtk.Button("Close")
        close.set_size_request(70, 30)
        valign.add(close)
        table.set_row_spacing(1, 3)
        table.attach(valign, 3, 4, 2, 3, gtk.FILL,
            gtk.FILL | gtk.EXPAND, 1, 1)

        halign2 = gtk.Alignment(0, 1, 0, 0)
        help = gtk.Button("Help")
        help.set_size_request(70, 30)
        halign2.add(help)
        table.set_row_spacing(3, 6)
        table.attach(halign2, 0, 1, 4, 5, gtk.FILL,
            gtk.FILL, 0, 0)

        ok = gtk.Button("OK")
```

```
        ok.set_size_request(70, 30)
        table.attach(ok, 3, 4, 4, 5, gtk.FILL,
            gtk.FILL, 0, 0);

        self.add(table)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

The code example shows, how we can create a similar window in PyGTK.

```
 table = gtk.Table(8, 4, False)
 table.set_col_spacings(3)
```

The example is based on the Table container. There will be 3 px space between columns.

```
 title = gtk.Label("Windows")

 halign = gtk.Alignment(0, 0, 0, 0)
 halign.add(title)

 table.attach(halign, 0, 1, 0, 1, gtk.FILL,
     gtk.FILL, 0, 0);
```

This code creates a label, that is aligned to the left. The label is placed in the first row of the Table container.

```
 wins = gtk.TextView()
 wins.set_editable(False)
 wins.modify_fg(gtk.STATE_NORMAL, gtk.gdk.Color(5140, 5140, 5140))
 wins.set_cursor_visible(False)
 table.attach(wins, 0, 2, 1, 3, gtk.FILL | gtk.EXPAND,
     gtk.FILL | gtk.EXPAND, 1, 1)
```

The text view widget spans two rows and two columns. We make the widget non editable and hide the cursor.

```
 valign = gtk.Alignment(0, 0, 0, 0)
 close = gtk.Button("Close")
 close.set_size_request(70, 30)
 valign.add(close)
 table.set_row_spacing(1, 3)
 table.attach(valign, 3, 4, 2, 3, gtk.FILL,
     gtk.FILL | gtk.EXPAND, 1, 1)
```

We put the close button next to the text view widget into the fourth column. (we count from zero) We add the button into the alignment widget, so that we can align it to the top.

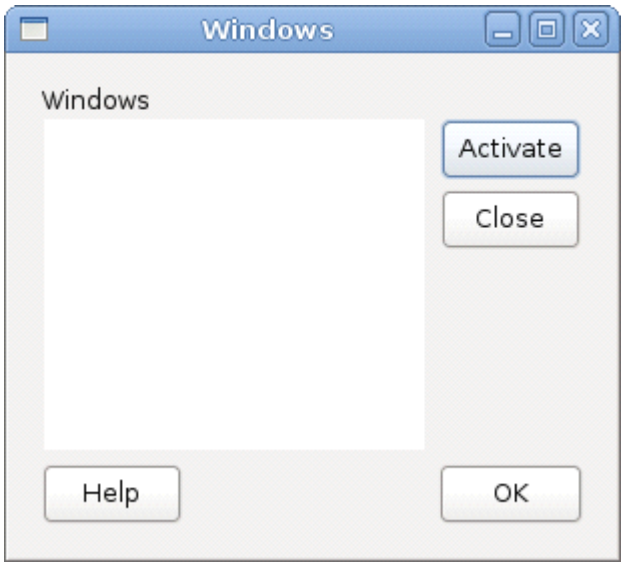


Figure: Windows

This chapter of the PyGTK programming tutorial was about layout management.

Menus in PyGTK

In this part of the PyGTK programming tutorial, we will work with menus.

A **menubar** is one of the most common parts of the GUI application. It is a group of commands located in various menus. While in console applications you have to remember all those arcane commands, here we have most of the commands grouped into logical parts. These are accepted standards that further reduce the amount of time spent to learn a new application.

Simple menu

In our first example, we will create a menubar with one file menu. The menu will have only one menu item. By selecting the item the application quits.

simplemenu.py

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

```
# ZetCode PyGTK tutorial
#
# This example shows a simple menu
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Simple menu")
        self.set_size_request(250, 200)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        mb = gtk.MenuBar()

        filemenu = gtk.Menu()
        filem = gtk.MenuItem("File")
        filem.set_submenu(filemenu)

        exit = gtk.MenuItem("Exit")
        exit.connect("activate", gtk.main_quit)
        filemenu.append(exit)

        mb.append(filem)

        vbox = gtk.VBox(False, 2)
        vbox.pack_start(mb, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

This is a small example with minimal menubar functionality.

```
 mb = gtk.MenuBar()
```

MenuBar widget is created.

```
 filemenu = gtk.Menu()
 filem = gtk.MenuItem("File")
 filem.set_submenu(filemenu)
```

Toplevel **MenuItem** is created.

```
exit = gtk.MenuItem("Exit")
exit.connect("activate", gtk.main_quit)
filemenu.append(exit)
```

Exit **MenuItem** is created and appended to the File **MenuItem**.

```
mb.append(filem)
```

Toplevel **MenuItem** is appended to the **MenuBar** widget.

```
vbox = gtk.VBox(False, 2)
vbox.pack_start(mb, False, False, 0)
```

Unlike in other toolkits, we have to take care of the layout management of the menubar ourselves. We put the menubar into a vertical box.

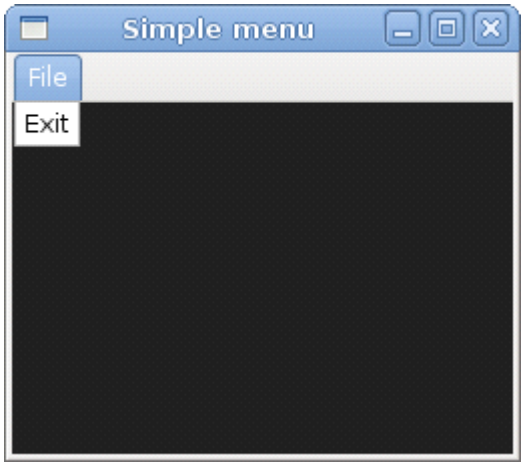


Figure: Simple menu

Image menu

In the next example, we will further explore the menus. We will add images and accelerators to our menu items. **Accelerators** are keyboard shortcuts for activating a menu item.

imagemenu.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows a menu with
# images, accelerators and a separator
#
# author: jan bodnar
# website: zetcode.com
```

```
# last edited: February 2009

import gtk


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Image menu")
        self.set_size_request(250, 200)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        mb = gtk.MenuBar()

        filemenu = gtk.Menu()
        filem = gtk.MenuItem("_File")
        filem.set_submenu(filemenu)

        agr = gtk.AccelGroup()
        self.add_accel_group(agr)

        newi = gtk.ImageMenuItem(gtk.STOCK_NEW, agr)
        key, mod = gtk.accelerator_parse("<Control>N")
        newi.add_accelerator("activate", agr, key,
            mod, gtk.ACCEL_VISIBLE)
        filemenu.append(newi)

        openm = gtk.ImageMenuItem(gtk.STOCK_OPEN, agr)
        key, mod = gtk.accelerator_parse("<Control>O")
        openm.add_accelerator("activate", agr, key,
            mod, gtk.ACCEL_VISIBLE)
        filemenu.append(openm)

        sep = gtk.SeparatorMenuItem()
        filemenu.append(sep)

        exit = gtk.ImageMenuItem(gtk.STOCK_QUIT, agr)
        key, mod = gtk.accelerator_parse("<Control>Q")
        exit.add_accelerator("activate", agr, key,
            mod, gtk.ACCEL_VISIBLE)

        exit.connect("activate", gtk.main_quit)

        filemenu.append(exit)

        mb.append(filem)

        vbox = gtk.VBox(False, 2)
        vbox.pack_start(mb, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()
```

```
PyApp()
gtk.main()
```

Our example shows a toplevel menu item with three sublevel menu items. Each of the menu items has a image and an accelerator. The accelerator for the quit menu item is active.

```
agr = gtk.AccelGroup()
self.add_accel_group(agr)
```

To work with accelerators, we create a global **AccelGroup** object. It will be used later.

```
newi = gtk.ImageMenuItem(gtk.STOCK_NEW, agr)
```

ImageMenuItem is created. The image comes from the stock of images.

```
key, mod = gtk.accelerator_parse("<Control>N")
```

The **gtk.accelerator_parse()** function parses the specified accelerator string and returns a 2-tuple containing the keyval and modifier mask corresponding to accelerator.

```
newi.add_accelerator("activate", agr, key,
    mod, gtk.ACCEL_VISIBLE)
```

This creates an Ctrl+Q accelerator for the exit menu item.

```
sep = gtk.SeparatorMenuItem()
filemenu.append(sep)
```

These lines create a separator. It is used to group menu items into logical groups.

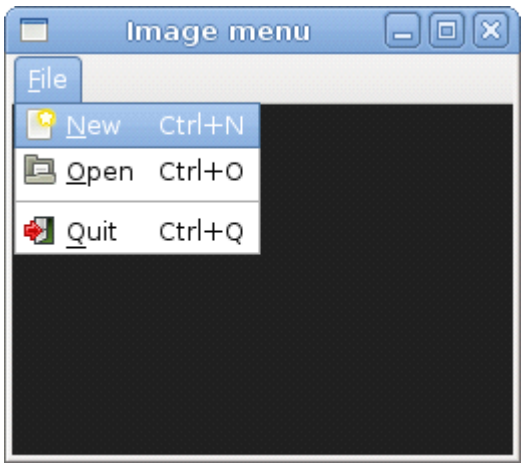


Figure: Image menu

CheckMenuItem

A CheckMenuItem is a menu item with a check box. It can be used to work with boolean properties.

checkmenuitem.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to
# use a CheckMenuItem
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Check menu item")
        self.set_size_request(250, 200)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        mb = gtk.MenuBar()

        filemenu = gtk.Menu()
        filem = gtk.MenuItem("File")
        filem.set_submenu(filemenu)

        viewmenu = gtk.Menu()
        view = gtk.MenuItem("View")
        view.set_submenu(viewmenu)

        stat = gtk.CheckMenuItem("View Statusbar")
        stat.set_active(True)
        stat.connect("activate", self.on_status_view)
        viewmenu.append(stat)

        exit = gtk.MenuItem("Exit")
        exit.connect("activate", gtk.main_quit)
        filemenu.append(exit)

        mb.append(filem)
```

```
        mb.append(view)

        self.statusbar = gtk.Statusbar()
        self.statusbar.push(1, "Ready")

        vbox = gtk.VBox(False, 2)
        vbox.pack_start(mb, False, False, 0)
        vbox.pack_start(gtk.Label(), True, False, 0)
        vbox.pack_start(self.statusbar, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


    def on_status_view(self, widget):
        if widget.active:
            self.statusbar.show()
        else:
            self.statusbar.hide()


PyApp()
gtk.main()
```

In our code example we show a check menu item. If the check box is activated, the statusbar widget is shown. If not, the statusbar is hidden.

```
stat = gtk.CheckMenuItem("View Statusbar")
```

CheckMenuItem widget is created.

```
stat.set_active(True)
```

The set_active() method checks/unchecks the check menu item.

```
if widget.active:
    self.statusbar.show()
else:
    self.statusbar.hide()
```

Depending on the active property of the CheckMenuItem, we show or hide the statusbar widget.

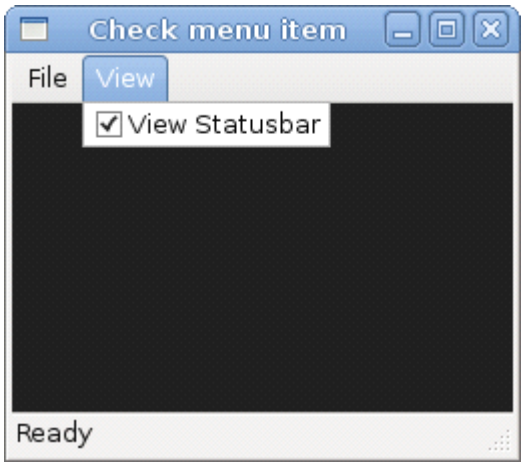


Figure: CheckMenuItem

Submenu

Our final example demonstrates how to create a submenu in PyGTK.

submenu.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows a submenu
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Submenu")
        self.set_size_request(250, 200)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        mb = gtk.MenuBar()

        filemenu = gtk.Menu()
        filem = gtk.MenuItem("File")
        filem.set_submenu(filemenu)

        mb.append(filem)
```

```
        imenu = gtk.Menu()

        importm = gtk.MenuItem("Import")
        importm.set_submenu(imenu)

        inews = gtk.MenuItem("Import news feed...")
        ibookmarks = gtk.MenuItem("Import bookmarks...")
        imail = gtk.MenuItem("Import mail...")

        imenu.append(inews)
        imenu.append(ibookmarks)
        imenu.append(imail)

        filemenu.append(importm)

        exit = gtk.MenuItem("Exit")
        exit.connect("activate", gtk.main_quit)
        filemenu.append(exit)

        vbox = gtk.VBox(False, 2)
        vbox.pack_start(mb, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

Submenu creation.

```
 imenu = gtk.Menu()
```

A submenu is a Menu.

```
 importm = gtk.MenuItem("Import")
 importm.set_submenu(imenu)
```

It is a submenu of a menu item, which belogs to toplevel file menu.

```
 inews = gtk.MenuItem("Import news feed...")
 ibookmarks = gtk.MenuItem("Import bookmarks...")
 imail = gtk.MenuItem("Import mail...")

 imenu.append(inews)
 imenu.append(ibookmarks)
 imenu.append(imail)
```

Submenu has it's own menu items.

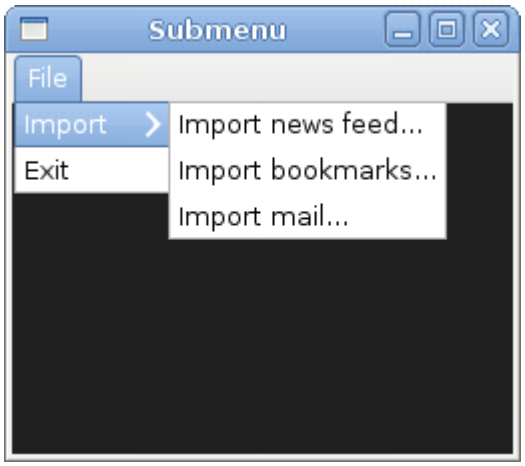


Figure: Submenu

In this chapter of the PyGTK programming library, we showed, how to work with menus.

Toolbars in PyGTK

In this part of the PyGTK programming tutorial, we will work with toolbars.

Menus group commands that we can use in application. Toolbars provide a quick access to the most frequently used commands.

Simple toolbar

Next we create a simple toolbar.

toolbar.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows a toolbar
# widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
```

```
class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Toolbar")
        self.set_size_request(250, 200)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        toolbar = gtk.Toolbar()
        toolbar.set_style(gtk.TOOLBAR_ICONS)

        newtb = gtk.ToolButton(gtk.STOCK_NEW)
        opentb = gtk.ToolButton(gtk.STOCK_OPEN)
        savetb = gtk.ToolButton(gtk.STOCK_SAVE)
        sep = gtk.SeparatorToolItem()
        quittb = gtk.ToolButton(gtk.STOCK_QUIT)

        toolbar.insert(newtb, 0)
        toolbar.insert(opentb, 1)
        toolbar.insert(savetb, 2)
        toolbar.insert(sep, 3)
        toolbar.insert(quittb, 4)

        quittb.connect("clicked", gtk.main_quit)

        vbox = gtk.VBox(False, 2)
        vbox.pack_start(toolbar, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

The example shows a toolbar and four tool buttons.

```
 toolbar = gtk.Toolbar()
```

A Toolbar widget is created.

```
 toolbar.set_style(gtk.TOOLBAR_ICONS)
```

On toolbar, we show only icons. No text.

```
 newtb = gtk.ToolButton(gtk.STOCK_NEW)
```

A ToolButton with an image from stock is created.

```
sep = gtk.SeparatorToolItem()
```

This is a separator. It can be used to group toolbar buttons into logical groups.

```
toolbar.insert(newtb, 0)
toolbar.insert(opentb, 1)
...
```

Toolbar buttons are inserted into the toolbar widget.

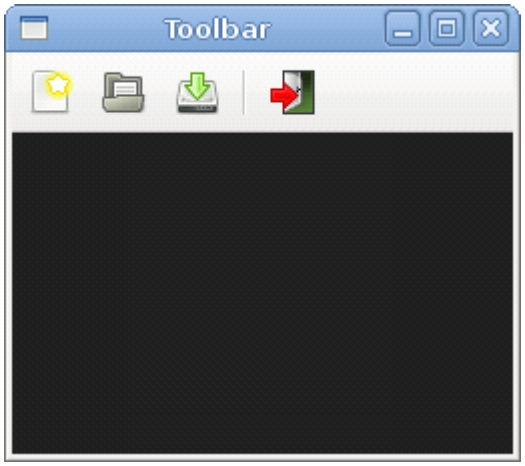


Figure: Toolbar

Toolbars

In the second example, we show two toolbars. Many applications have more than one toolbar. We show, how we can do it in PyGTK.

toolbars.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows two toolbars
# in the application window
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):
```

```
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Toolbars")
        self.set_size_request(350, 300)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        upper = gtk.Toolbar()
        upper.set_style(gtk.TOOLBAR_ICONS)

        newtb = gtk.ToolButton(gtk.STOCK_NEW)
        opentb = gtk.ToolButton(gtk.STOCK_OPEN)
        savetb = gtk.ToolButton(gtk.STOCK_SAVE)

        upper.insert(newtb, 0)
        upper.insert(opentb, 1)
        upper.insert(savetb, 2)

        lower = gtk.Toolbar()
        lower.set_style(gtk.TOOLBAR_ICONS)

        quittb = gtk.ToolButton(gtk.STOCK_QUIT)
        quittb.connect("clicked", gtk.main_quit)
        lower.insert(quittb, 0)

        vbox = gtk.VBox(False, 0)
        vbox.pack_start(upper, False, False, 0)
        vbox.pack_start(lower, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


PyApp()
gtk.main()
```

Our applications shows two toolbars.

```
upper = gtk.Toolbar()
...
lower = gtk.Toolbar()
```

We create two Toolbar widgets.

```
upper.insert(newtb, 0)
...
lower.insert(quittb, 0)
```

Each of them has it's own tool buttons.

```
vbox = gtk.VBox(False, 0)
vbox.pack_start(upper, False, False, 0)
```

```
vbox.pack_start(lower, False, False, 0)
```

Toolbars are packed into the vertical box, one after the other.

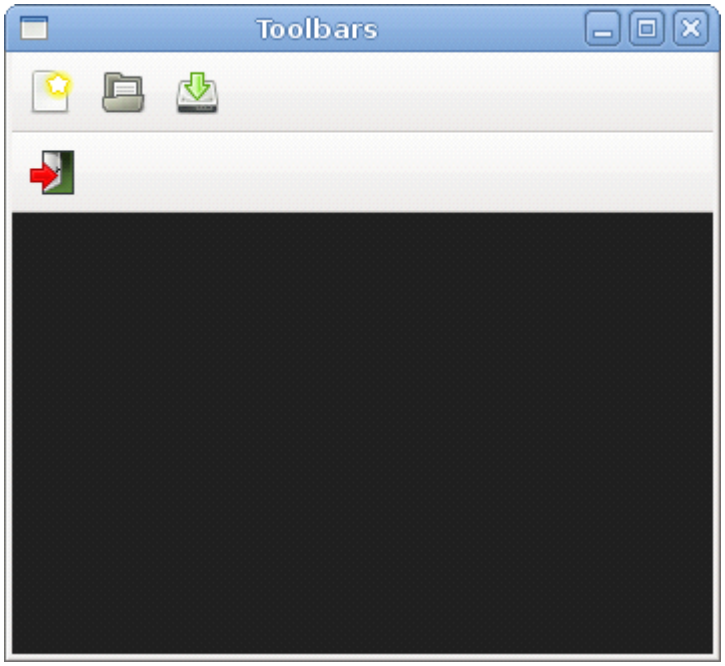


Figure: Toolbars

Undo redo

The following example demonstrates, how we can inactivate toolbar buttons on the toolbar. It is a common practise in GUI programming. For example the save button. If we save all changes of our document to the disk, the save button is inactivated in most text editors. This way the application indicates to the user, that all changes are already saved.

undoredo.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to
# activate/deactivate a ToolButton
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
```

```
class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Toolbar")
        self.set_size_request(250, 200)
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(6400, 6400, 6440))
        self.set_position(gtk.WIN_POS_CENTER)

        self.count = 2

        toolbar = gtk.Toolbar()
        toolbar.set_style(gtk.TOOLBAR_ICONS)

        self.undo = gtk.ToolButton(gtk.STOCK_UNDO)
        self.redo = gtk.ToolButton(gtk.STOCK_REDO)
        sep = gtk.SeparatorToolItem()
        quit = gtk.ToolButton(gtk.STOCK_QUIT)

        toolbar.insert(self.undo, 0)
        toolbar.insert(self.redo, 1)
        toolbar.insert(sep, 2)
        toolbar.insert(quit, 3)

        self.undo.connect("clicked", self.on_undo)
        self.redo.connect("clicked", self.on_redo)
        quit.connect("clicked", gtk.main_quit)

        vbox = gtk.VBox(False, 2)
        vbox.pack_start(toolbar, False, False, 0)

        self.add(vbox)

        self.connect("destroy", gtk.main_quit)
        self.show_all()

    def on_undo(self, widget):
        self.count = self.count - 1

        if self.count <= 0:
            self.undo.set_sensitive(False)
            self.redo.set_sensitive(True)


    def on_redo(self, widget):
        self.count = self.count + 1

        if self.count >= 5:
            self.redo.set_sensitive(False)
            self.undo.set_sensitive(True)

PyApp()
gtk.main()
```

Our example creates undo and redo buttons from the PyGTK stock resources. After several clicks each of the buttons is inactivated. The buttons are grayed out.

```
self.count = 2
```

The **self.count** variable decides, which button is activated and deactivated.

```
self.undo = gtk.ToolButton(gtk.STOCK_UNDO)
self.redo = gtk.ToolButton(gtk.STOCK_REDO)
```

We have two tool buttons. Undo and redo tool buttons. Images come from the stock resources.

```
self.undo.connect("clicked", self.on_undo)
self.redo.connect("clicked", self.on_redo)
```

We plug a method for the **clicked** signal for both tool buttons.

```
if self.count <= 0:
    self.undo.set_sensitive(False)
    self.redo.set_sensitive(True)
```

To activate a widget, we use the **set_sensitive()** method.

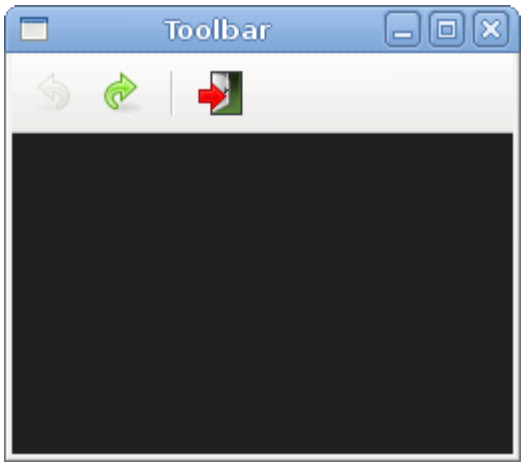


Figure: Undo redo

In this chapter of the PyGTK programming library, we mentioned toolbars.

Signals & events in PyGTK

In this part of the PyGTK programming tutorial, we will talk about signals & events.

All GUI applications are event driven. PyGTK applications are no exception. The applications start a main loop with the **gtk.main()** call, which continuously checks for newly generated events. If there is no event, the application waits and does nothing.

Events are messages from the X server to the application. When we click on a button widget, the clicked signal will be **emitted**. There are signals that all widgets inherit, such as destroy, and there are signals that are widget specific, such as toggled on a toggle button.

Programmers use signal handlers to react to various signals. These handlers are called **callbacks** among GTK programmers.

handler_id = button.connect("clicked", self.on_clicked)

Here we use the **connect()** method of the GObject class, (GtkButton is a GObject), to connect a callback **on_clicked()** to a signal called **clicked**.

The connect() method returns a handler id, which is used to uniquely identify the callback method. The id can be used with the following methods:

```
def disconnect(handler_id)
def handler_disconnect(handler_id)
def handler_is_connected(handler_id)
def handler_block(handler_id)
def handler_unblock(handler_id)
```

These methods enable to disconnect a handler from an GObject, or block/unblock it.

Signals vs events

There is generally a lot of confusion about the difference between the two.

Signals and events are two different things. An event is an almost one-to-one mapping of window system events. Key press, window resizement or button press are typical window system events. Window system events

are reported to the application main loop. Gdk interprets the window system events and passes them along via signals.

A signal is nothing other than a callback mechanism. If one object wants to be notified about an other object's action or state change, it registers a callback. When the object emits a signal, it looks in the list of callbacks which have been registered with it and calls the callback(s) for the specific signal. It can optionally send some predefined data with it.

Signals are a general purpose notification framework. They are not used only for notifications about UI changes. They can be used for notifications about application state changes. Signals are general, powerful, their usage is very broad. Any GObject can emit and receive a signal. A type may have one or more signals, each of which may have an argument list and return value. Handlers can then be connected to instances of the type. When the signal is emitted on an instance, each of the connected handlers will be called.

The only connection between signals and events is that signals are used to send notifications about events from the X server.

Signals are a feature of gtk.Object and its subclasses, events are a Gdk/Xlib concept.

Simple example

The next example shows, how we react to two basic signals.

quitbutton.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# The example shows how to work with
# destroy and clicked signals
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk

class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()
```

```
        self.set_title("Quit Button")
        self.set_size_request(250, 200)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", self.on_destroy)

        fixed = gtk.Fixed()

        quit = gtk.Button("Quit")
        quit.connect("clicked", self.on_clicked)
        quit.set_size_request(80, 35)

        fixed.put(quit, 50, 50)

        self.add(fixed)
        self.show_all()

    def on_destroy(self, widget):
        gtk.main_quit()

    def on_clicked(self, widget):
        gtk.main_quit()


PyApp()
gtk.main()
```

The destroy signal is triggered, when we close the window. By default, the application does not quit, when we click on the close button in the titlebar.

```
self.connect("destroy", self.on_destroy)
```

The **connect()** method plugs the **on_destroy()** method to the **destroy** signal.

```
quit.connect("clicked", self.on_clicked)
```

Pressing the quit button, the **clicked** signal is triggered. When we click on the quit button, we call the **on_clicked()** method.

```
def on_destroy(self, widget):
    gtk.main_quit()
```

In the **on_destroy()** method, we react to the **destroy** signal. We call the **gtk.main_quit()** method, which terminates the application.

```
def on_clicked(self, widget):
    gtk.main_quit()
```

Here is the **on_clicked()** method. It takes two parameters. The widget parameter is the object, which triggered this signal. In our case it is the quit button. Different objects send different signals. Signals and

the parameters sent to methods can be found in the reference manual of the PyGTK library. pygtk.org/docs/pygtk/index.html

Creating a custom signal

In the following code example, we create and send a custom singal.

customsignal.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to create
# and send a custom singal
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gobject


class Sender(gobject.GObject):
    def __init__(self):
        self.__gobject_init__()


gobject.type_register(Sender)
gobject.signal_new("z_signal", Sender, gobject.SIGNAL_RUN_FIRST,
                   gobject.TYPE_NONE, ())


class Receiver(gobject.GObject):
    def __init__(self, sender):
        self.__gobject_init__()

        sender.connect('z_signal', self.report_signal)

    def report_signal(self, sender):
        print "Receiver reacts to z_signal"


def user_callback(object):
    print "user callback reacts to z_signal"

if __name__ == '__main__':

    sender = Sender()
    receiver = Receiver(sender)

    sender.connect("z_signal", user_callback)
    sender.emit("z_signal")
```

We create two GObjects. Sender and receiver objects. The sender emits a signal, which is received by the receiver object. We also plug a callback to the signal.

```
class Sender(gobject.GObject):
    def __init__(self):
        self.__gobject_init__()
```

This is a sender object. It is created with a default constructor.

```
gobject.type_register(Sender)
gobject.signal_new("z_signal", Sender, gobject.SIGNAL_RUN_FIRST,
                   gobject.TYPE_NONE, ())
```

We register a new object and a new signal. The signal_new() function registers a signal called z_signal for the Sender object. The SIGNAL_RUN_FIRST parameter means that the default handler of the object that receives the signal is called as first. The last two parameters are the return value type and parameter types. In our example we do not return any value and send no parameters.

```
sender.connect('z_signal', self.report_signal)
```

The receiver listens for the z_signal.

```
sender = Sender()
receiver = Receiver(sender)
```

Sender and receiver objects are instantiated. The receiver takes a sender as a parameter, so that it can listen to its signals.

```
sender.connect("z_signal", user_callback)
```

Here we plug the signal to the user callback.

```
sender.emit("z_signal")
```

The z_signal is being emitted.

```
class Sender(gobject.GObject):

    __gsignals__ = {
        'z_signal': (gobject.SIGNAL_RUN_LAST, gobject.TYPE_NONE, ()),
    }

    def __init__(self):
        self.__gobject_init__()

gobject.type_register(Sender)
```

We can also use the __gsignals__ class attribute to register a new singal.

Predefined signal handlers

Objects in PyGTK may have predefined signal handlers. These handlers begin with do_*. For example do_expose(), do_show() or do_clicked().

move.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example overrides predefined
# do_configure_event() signal handler
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import gobject

class PyApp(gtk.Window):
    __gsignals__ = {
        "configure-event" : "override"
        }

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(200, 150)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        self.show_all()

    def do_configure_event(self, event):

        title = "%s, %s" % (event.x, event.y)
        self.set_title(title)
        gtk.Window.do_configure_event(self, event)


PyApp()
gtk.main()
```

When we move or resize a window, the X server sends configure events. These are then transformed into configure-event signals.

In our code example, we display the x, y coordinates of the top-left corner of the window in the titlebar. We could simply connect a signal handler to the **configure-event** signal. But we take a different strategy. We override the default class handler, where we implement the logic needed.

```
__gsignals__ = {
    "configure-event" : "override"
    }
```

This tells, that we are going to override the default **on_configure_event()** method.

```
def do_configure_event(self, event):

    title = "%s, %s" % (event.x, event.y)
    self.set_title(title)
    gtk.Window.do_configure_event(self, event)
```

Here we override the predefined **do_configure_event()** method. We set the x, y coordinates of the window to the title of the window. Also note the last line. It explicitly calls the superclass **do_configure_event()** method. This is beacuse it does some important job. Try to comment this line to see what happens. Resizing of windows would not work correctly. If we override a default handler, we may or may not call the superclass method. In our case we have to.

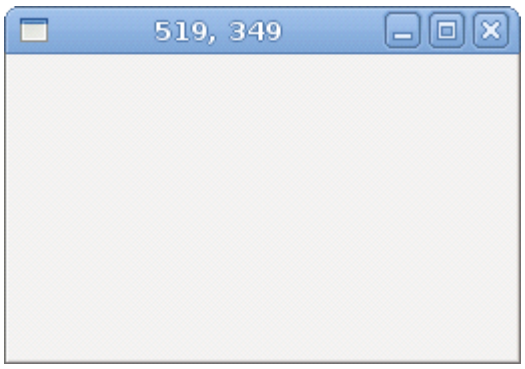


Figure: Configure singal

Signals of a button

The following example shows various button signals.

buttonsignals.py

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

```
# ZetCode PyGTK tutorial
#
# This program shows various signals
# of a button widget
# It emits a button-release-event which
# triggers a released singal
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Signals")
        self.set_size_request(250, 200)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)

        fixed = gtk.Fixed()

        self.quit = gtk.Button("Quit")

        self.quit.connect("pressed", self.on_pressed)
        self.quit.connect("released", self.on_released)
        self.quit.connect("clicked", self.on_clicked)

        self.quit.set_size_request(80, 35)

        fixed.put(self.quit, 50, 50)

        self.add(fixed)
        self.show_all()
        self.emit_signal()

    def emit_signal(self):

        event = gtk.gdk.Event(gtk.gdk.BUTTON_RELEASE)
        event.button = 1
        event.window = self.quit.window
        event.send_event = True

        self.quit.emit("button-release-event", event)


    def on_clicked(self, widget):
        print "clicked"

    def on_released(self, widget):
        print "released"

    def on_pressed(self, widget):
        print "pressed"
```

```
PyApp()
gtk.main()
```

A button can emit more than just one type of signal. We work with three of them. The clicked, pressed and released signals. We also show, how an event signal triggers another signal.

```
 self.quit.connect("pressed", self.on_pressed)
 self.quit.connect("released", self.on_released)
 self.quit.connect("clicked", self.on_clicked)
```

We register callbacks for all three signals.

```
 self.emit_signal()
```

Upon the start of the application, we emit a specific signal.

```
 def emit_signal(self):

     event = gtk.gdk.Event(gtk.gdk.BUTTON_RELEASE)
     event.button = 1
     event.window = self.quit.window
     event.send_event = True

     self.quit.emit("button-release-event", event)
```

We emit the button-release-event signal. It takes an Event object as a parameter. After the application starts, we should see "released" text in our console window. When we click on the button, all three signals are triggered.

Blocking an event handler

We can block a signal handler. The next example shows this.

block.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to block/unblock
# a signal handler
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009
```

```
import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Blocking a callback")
        self.set_size_request(250, 180)
        self.set_position(gtk.WIN_POS_CENTER)

        fixed = gtk.Fixed()
        button = gtk.Button("Click")
        button.set_size_request(80, 35)
        self.id = button.connect("clicked", self.on_clicked)
        fixed.put(button, 30, 50)

        check = gtk.CheckButton("Connect")
        check.set_active(True)
        check.connect("clicked", self.toggle_blocking, button)
        fixed.put(check, 130, 50)


        self.connect("destroy", gtk.main_quit)

        self.add(fixed)
        self.show_all()

    def on_clicked(self, widget):
        print "clicked"

    def toggle_blocking(self, checkbox, button):
        if checkbox.get_active():
           button.handler_unblock(self.id)
        else:
           button.handler_block(self.id)

PyApp()
gtk.main()
```

In the code example, we have a button and a check box. We show "clicked" text in the console, when we click on the button and the check box is active. The check box blocks/unblocks a handler method from the button `clicked` signal.

```
self.id = button.connect("clicked", self.on_clicked)
```

The `connect()` method returns a handler id. This id is used to block and unblock the handler.

```
def toggle_blocking(self, checkbox, button):
    if checkbox.get_active():
       button.handler_unblock(self.id)
    else:
       button.handler_block(self.id)
```

These lines block and unblock the callback with the appropriate methods.

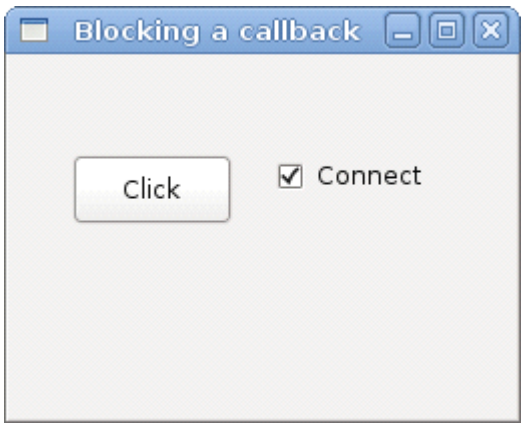


Figure: Blocking a callback

In this chapter of the PyGTK tutorial, we worked with signals.

Widgets in PyGTK

In this part of the PyGTK programming tutorial, we will introduce some PyGTK widgets.

Widgets are basic building blocks of a GUI application. Over the years, several widgets became a standard in all toolkits on all OS platforms. For example a button, a check box or a scroll bar. The PyGTK toolkit's philosophy is to keep the number of widgets at a minimum level. More specialized widgets are created as custom PyGTK widgets.

Label

The **Label** widget displays a limited amount of read-only text.

label.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the Label widget
#
# author: jan bodnar
```

```
# website: zetcode.com
# last edited: February 2009

import gtk

lyrics = """Meet you downstairs in the bar and heard
your rolled up sleeves and your skull t-shirt
You say why did you do it with him today?
and sniff me out like I was Tanqueray

cause you're my fella, my guy
hand me your stella and fly
by the time I'm out the door
you tear men down like Roger Moore

I cheated myself
like I knew I would
I told ya, I was trouble
you know that I'm no good"""


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_position(gtk.WIN_POS_CENTER)
        self.set_border_width(8)
        self.connect("destroy", gtk.main_quit)
        self.set_title("You know I'm no Good")

        label = gtk.Label(lyrics)
        self.add(label)
        self.show_all()


PyApp()
gtk.main()
```

The code example shows some lyrics on the window.

```
lyrics = """Meet you downstairs in the bar and heard
your rolled up sleeves and your skull t-shirt
..."""
```

This is the text that we display.

```
 self.set_border_width(8)
```

The Label is surrounded by some empty space.

```
 label = gtk.Label(lyrics)
 self.add(label)
```

The Label widget is created and added to the window.

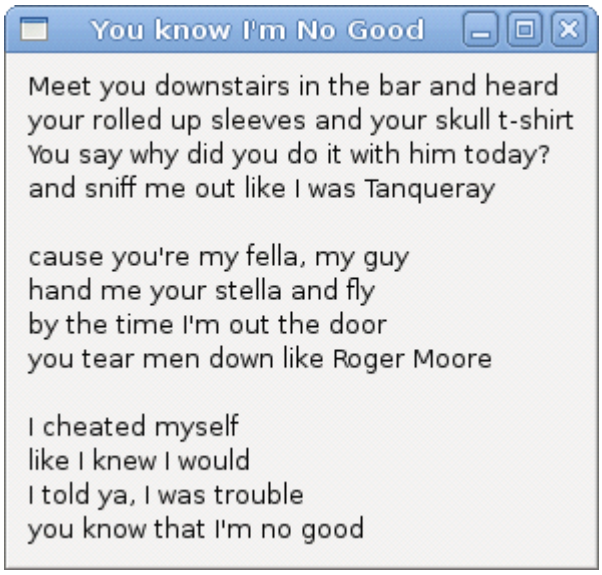


Figure: Label Widget

CheckButton

CheckButton is a widget, that has two states. On and Off. The On state is visualised by a check mark. It is used to denote some boolean property.

checkbutton.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the CheckButton widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()
        self.set_title("Check Button")
        self.set_position(gtk.WIN_POS_CENTER)
        self.set_default_size(250, 200)

        fixed = gtk.Fixed()
        button = gtk.CheckButton("Show title")
        button.set_active(True)
        button.unset_flags(gtk.CAN_FOCUS)
```

```
        button.connect("clicked", self.on_clicked)

        fixed.put(button, 50, 50)

        self.connect("destroy", gtk.main_quit)
        self.add(fixed)
        self.show_all()

    def on_clicked(self, widget):
        if widget.get_active():
            self.set_title("Check Button")
        else:
           self.set_title("")

PyApp()
gtk.main()
```

We will display a title in the titlebar of the window, depending on the state of the CheckButton.

```
button = gtk.CheckButton("Show title")
```

CheckButton widget is created.

```
button.set_active(True)
```

The title is visible by default, so we check the check button by default.

```
if widget.get_active():
    self.set_title("Check Button")
else:
    self.set_title("")
```

If the CheckButton is checked we show the title. Otherwise we put empty text in the titlebar.

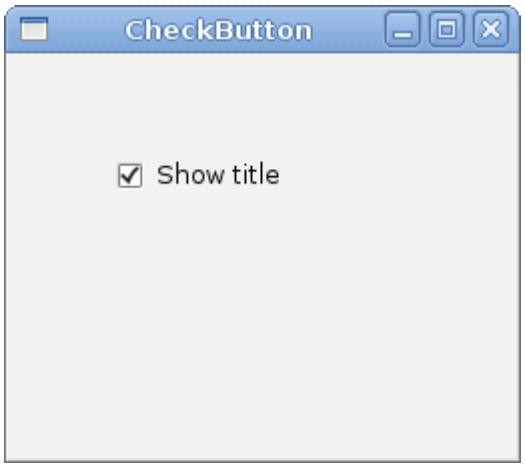


Figure: CheckButton

ComboBox

ComboBox is a widget that allows the user to choose from a list of options.

combobox.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the ComboBox widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk

class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("ComboBox")
        self.set_default_size(250, 200)
        self.set_position(gtk.WIN_POS_CENTER)

        cb = gtk.combo_box_new_text()
        cb.connect("changed", self.on_changed)

        cb.append_text('Ubuntu')
        cb.append_text('Mandriva')
        cb.append_text('Redhat')
        cb.append_text('Gentoo')
        cb.append_text('Mint')

        fixed = gtk.Fixed()
        fixed.put(cb, 50, 30)
        self.label = gtk.Label("-")
        fixed.put(self.label, 50, 140)
        self.add(fixed)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


    def on_changed(self, widget):
        self.label.set_label(widget.get_active_text())

PyApp()
gtk.main()
```

The example shows a combo box and a label. The combo box has a list of six options. These are the names of Linux Distros. The label widget shows the selected option from the combo box.

```
cb = gtk.combo_box_new_text()
```

The gtk.combo_box_new_text() function is a convenience function that constructs a new text combo box. It is a ComboBox just displaying strings.

```
cb.append_text('Ubuntu')
cb.append_text('Mandriva')
cb.append_text('Redhat')
cb.append_text('Gentoo')
cb.append_text('Mint')
```

The ComboBox is filled with textual data.

```
self.label.set_label(widget.get_active_text())
```

Inside the on_changed() method, we get the selected text out of the combo box and set it to the label.

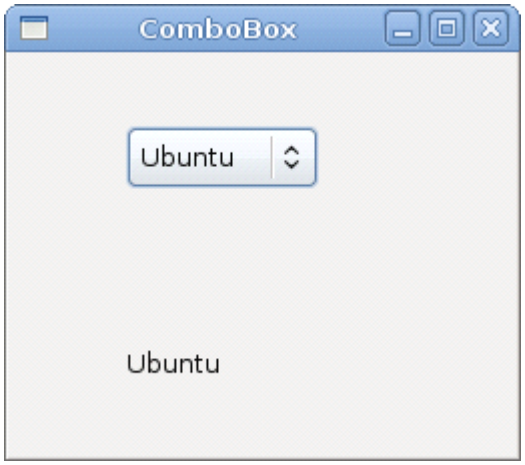


Figure: ComboBox

Image

The next example introduces the Image widget. This widget displays pictures.

image.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the Image widget
#
# author: jan bodnar
# website: zetcode.com
```

```
# last edited: February 2009

import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Red Rock")
        self.set_position(gtk.WIN_POS_CENTER)
        self.set_border_width(2)

        image = gtk.Image()
        image.set_from_file("redrock.png")

        self.connect("destroy", gtk.main_quit)
        self.add(image)
        self.show_all()

PyApp()
gtk.main()
```

We show the Red Rock castle in the window.

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

Image widget is created.

```
image.set_from_file("redrock.png")
```

We set a png image to the Image widget. The picture is loaded from the file on the disk.

Figure: Image

In this chapter, we showed the first pack of basic widgets of the PyGTK programming library.

Widgets II in PyGTK

In this part of the PyGTK programming tutorial, we continue introducing PyGTK widgets.

Entry

The **Entry** is a single line text entry field. This widget is used to enter textual data.

entry.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the Entry widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Entry")
        self.set_size_request(250, 200)
        self.set_position(gtk.WIN_POS_CENTER)

        fixed = gtk.Fixed()

        self.label = gtk.Label("...")
        fixed.put(self.label, 60, 40)

        entry = gtk.Entry()
        entry.add_events(gtk.gdk.KEY_RELEASE_MASK)
        fixed.put(entry, 60, 100)
```

```
        entry.connect("key-release-event", self.on_key_release)

        self.connect("destroy", gtk.main_quit)
        self.add(fixed)
        self.show_all()

    def on_key_release(self, widget, event):
        self.label.set_text(widget.get_text())

PyApp()
gtk.main()
```

This example shows an entry widget and a label. The text that we key in the entry is displayed immediately in the label control.

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

Entry widget is created.

```
entry.connect("key-release-event", self.on_key_release)
```

If the text in the Entry widget is changed, we call the on_key_release() method.

```
def on_key_release(self, widget, event):
    self.label.set_text(widget.get_text())
```

We get the text from the Entry widget and set it to the label.

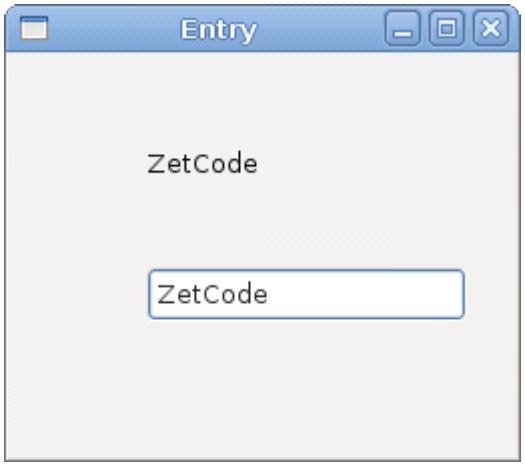


Figure: Entry Widget

HScale

The HScale is It is a horizontal slider, that lets the user graphically select a value by sliding a knob within a bounded interval. Our example will show a volume control.

hscale.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the HScale widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import sys


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Scale")
        self.set_size_request(260, 150)
        self.set_position(gtk.WIN_POS_CENTER)

        scale = gtk.HScale()
        scale.set_range(0, 100)
        scale.set_increments(1, 10)
        scale.set_digits(0)
        scale.set_size_request(160, 35)
        scale.connect("value-changed", self.on_changed)

        self.load_pixbufs()

        self.image = gtk.Image()
        self.image.set_from_pixbuf(self.mutp)

        fix = gtk.Fixed()
        fix.put(scale, 20, 40)
        fix.put(self.image, 219, 50)

        self.add(fix)

        self.connect("destroy", lambda w: gtk.main_quit())
        self.show_all()

    def load_pixbufs(self):
```

```
        try:
            self.mutp = gtk.gdk.pixbuf_new_from_file("mute.png")
            self.minp = gtk.gdk.pixbuf_new_from_file("min.png")
            self.medp = gtk.gdk.pixbuf_new_from_file("med.png")
            self.maxp = gtk.gdk.pixbuf_new_from_file("max.png")

        except Exception, e:
            print "Error reading Pixbufs"
            print e.message
            sys.exit(1)


    def on_changed(self, widget):
        val = widget.get_value()

        if val == 0:
            self.image.set_from_pixbuf(self.mutp)
        elif val > 0 and val <= 30:
            self.image.set_from_pixbuf(self.minp)
        elif val > 30 and val < 80:
            self.image.set_from_pixbuf(self.medp)
        else:
            self.image.set_from_pixbuf(self.maxp)


PyApp()
gtk.main()
```

In the example above, we have **HScale** and **Image** widgets. By dragging the scale we change the image on the **Image** widget.

```
scale = gtk.HScale()
```

HScale widget is created.

```
scale.set_range(0, 100)
```

We set the lower and upper boundaries of the scale.

```
scale.set_increments(1, 10)
```

The **set_increments()** method sets the step and page sizes for the range.

```
scale.set_digits(0)
```

We want to have integer values on the scale, so we set the number of decimal places to zero.

```
if val == 0:
    self.image.set_from_pixbuf(self.mutp)
elif val > 0 and val <= 30:
    self.image.set_from_pixbuf(self.minp)
```

```
 elif val > 30 and val < 80:
     self.image.set_from_pixbuf(self.medp)
 else:
     self.image.set_from_pixbuf(self.maxp)
```

Depending on the obtained value, we change the picture in the image widget.

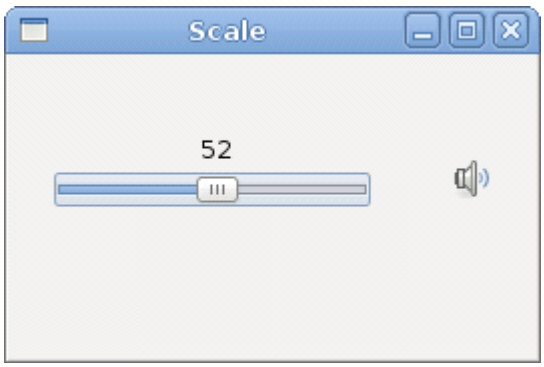


Figure: HScale Widget

ToggleButton

ToggleButton is a button that has two states. Pressed and not pressed. You toggle between these two states by clicking on it. There are situations where this functionality fits well.

togglebuttons.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the ToggleButton widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk

class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.color = [0, 0, 0]

        self.set_title("ToggleButtons")
        self.resize(350, 240)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)
```

```
        red = gtk.ToggleButton("Red")
        red.set_size_request(80, 35)
        red.connect("clicked", self.onred)
        green = gtk.ToggleButton("Green")
        green.set_size_request(80, 35)
        green.connect("clicked", self.ongreen)
        blue = gtk.ToggleButton("Blue")
        blue.set_size_request(80, 35)
        blue.connect("clicked", self.onblue)

        self.darea = gtk.DrawingArea()
        self.darea.set_size_request(150, 150)
        self.darea.modify_bg(gtk.STATE_NORMAL,
gtk.gdk.color_parse("black"))

        fixed = gtk.Fixed()
        fixed.put(red, 30, 30)
        fixed.put(green, 30, 80)
        fixed.put(blue, 30, 130)
        fixed.put(self.darea, 150, 30)

        self.add(fixed)

        self.show_all()

    def onred(self, widget):
        if widget.get_active():
            self.color[0] = 65535
        else: self.color[0] = 0

        self.darea.modify_bg(gtk.STATE_NORMAL,
gtk.gdk.Color(self.color[0],
            self.color[1], self.color[2]))

    def ongreen(self, widget):
        if (widget.get_active()):
            self.color[1] = 65535
        else: self.color[1] = 0

        self.darea.modify_bg(gtk.STATE_NORMAL,
gtk.gdk.Color(self.color[0],
            self.color[1], self.color[2]))

    def onblue(self, widget):
        if (widget.get_active()):
            self.color[2] = 65535
        else: self.color[2] = 0

        self.darea.modify_bg(gtk.STATE_NORMAL,
gtk.gdk.Color(self.color[0],
            self.color[1], self.color[2]))


PyApp()
gtk.main()
```

In our example, we show three toggle buttons and a DrawingArea. We set the background color of the area to black. The togglebuttons will toggle the red, green and blue parts of the color value. The background color will depend on which togglebuttons we have pressed.

```
self.color = [0, 0, 0]
```

This is the color value that is going to be updated with the toggle buttons.

```
red = gtk.ToggleButton("Red")
red.set_size_request(80, 35)
red.connect("clicked", self.onred)
```

The ToggleButton widget is created. We set it's size to 80x35 pixels. Each of the toggle buttons has it's own handler method.

```
self.darea = gtk.DrawingArea()
self.darea.set_size_request(150, 150)
self.darea.modify_bg(gtk.STATE_NORMAL, gtk.gdk.color_parse("black"))
```

The DrawingArea widget is the widget, that displays the color, mixed by the toggle buttons. At start, it shows black color.

```
if widget.get_active():
    self.color[0] = 65535
else: self.color[0] = 0
```

If the toggle button is pressed, we change the R, G or B part of the color accordingly.

```
self.darea.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(self.color[0],
          self.color[1], self.color[2]))
```

We update the color of the DrawingArea widget.

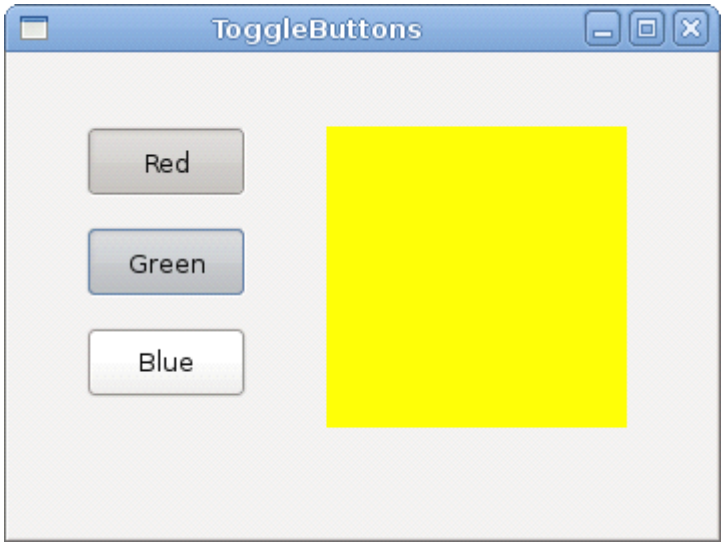


Figure: ToggleButton widget

Calendar

Our final widget is the Calendar widget. It is used to work with dates.

calendar.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the Calendar widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Calendar")
        self.set_size_request(300, 270)
        self.set_position(gtk.WIN_POS_CENTER)
        self.set_border_width(2)

        self.label = gtk.Label("...")

        calendar = gtk.Calendar()
        calendar.connect("day_selected", self.on_day_selected)
```

```
        fix = gtk.Fixed()
        fix.put(calendar, 20, 20)
        fix.put(self.label, 40, 230)

        self.add(fix)

        self.connect("destroy", gtk.main_quit)
        self.show_all()

    def on_day_selected(self, widget):
        (year, month, day) = widget.get_date()
        self.label.set_label(str(month) + "/" + str(day) + "/" + str(year))


PyApp()
gtk.main()
```

We have the **Calendar** widget and a **Label**. The selected day from the calendar is shown in the label.

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

Calendar widget is created.

```
(year, month, day) = widget.get_date()
self.label.set_label(str(month) + "/" + str(day) + "/" + str(year))
```

In the **on_day_selected()** method we retrieve the currently selected date, and update the label.

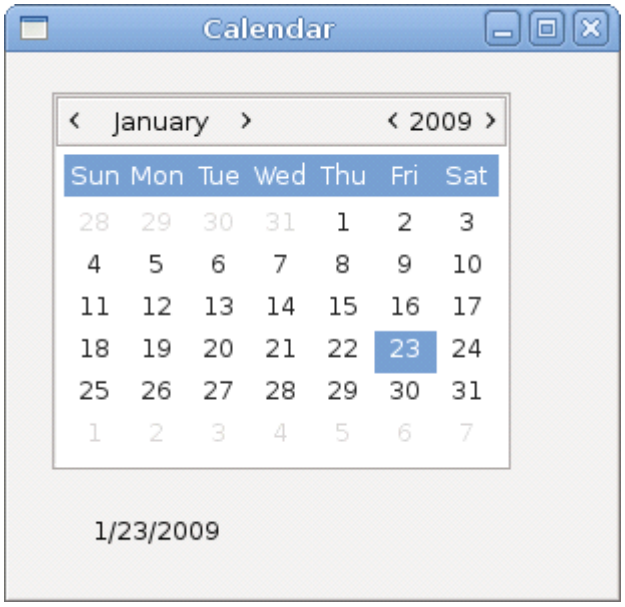


Figure: Calendar

In this chapter of the PyGTK tutorial, we finished talking about the PyGTK widgets.

Advanced widgets in PyGTK

In this part of the PyGTK programming tutorial, we will introduce some more advanced widgets in PyGTK.

IconView

The IconView is a widget which displays a list of icons in a grid.

iconview.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the IconView widget.
# It shows the contents of the currently selected
# directory on the disk.
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import os

COL_PATH = 0
COL_PIXBUF = 1
COL_IS_DIRECTORY = 2


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(650, 400)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)
        self.set_title("IconView")

        self.current_directory = '/'

        vbox = gtk.VBox(False, 0);
```

```
    toolbar = gtk.Toolbar()
    vbox.pack_start(toolbar, False, False, 0)

    self.upButton = gtk.ToolButton(gtk.STOCK_GO_UP);
    self.upButton.set_is_important(True)
    self.upButton.set_sensitive(False)
    toolbar.insert(self.upButton, -1)

    homeButton = gtk.ToolButton(gtk.STOCK_HOME)
    homeButton.set_is_important(True)
    toolbar.insert(homeButton, -1)

    self.fileIcon = self.get_icon(gtk.STOCK_FILE)
    self.dirIcon = self.get_icon(gtk.STOCK_OPEN)

    sw = gtk.ScrolledWindow()
    sw.set_shadow_type(gtk.SHADOW_ETCHED_IN)
    sw.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
    vbox.pack_start(sw, True, True, 0)

    self.store = self.create_store()
    self.fill_store()

    iconView = gtk.IconView(self.store)
    iconView.set_selection_mode(gtk.SELECTION_MULTIPLE)

    self.upButton.connect("clicked", self.on_up_clicked)
    homeButton.connect("clicked", self.on_home_clicked)

    iconView.set_text_column(COL_PATH)
    iconView.set_pixbuf_column(COL_PIXBUF)

    iconView.connect("item-activated", self.on_item_activated)
    sw.add(iconView)
    iconView.grab_focus()

    self.add(vbox)
    self.show_all()

def get_icon(self, name):
    theme = gtk.icon_theme_get_default()
    return theme.load_icon(name, 48, 0)


def create_store(self):
    store = gtk.ListStore(str, gtk.gdk.Pixbuf, bool)
    store.set_sort_column_id(COL_PATH, gtk.SORT_ASCENDING)
    return store


def fill_store(self):
    self.store.clear()

    if self.current_directory == None:
        return

    for fl in os.listdir(self.current_directory):
```

```
            if not fl[0] == '.':
                if os.path.isdir(os.path.join(self.current_directory,
fl)):
                    self.store.append([fl, self.dirIcon, True])
                else:
                    self.store.append([fl, self.fileIcon, False])



    def on_home_clicked(self, widget):
        self.current_directory =
os.path.realpath(os.path.expanduser('~'))
        self.fill_store()
        self.upButton.set_sensitive(True)


    def on_item_activated(self, widget, item):

        model = widget.get_model()
        path = model[item][COL_PATH]
        isDir = model[item][COL_IS_DIRECTORY]

        if not isDir:
            return

        self.current_directory = self.current_directory + os.path.sep +
path
        self.fill_store()
        self.upButton.set_sensitive(True)


    def on_up_clicked(self, widget):
        self.current_directory = os.path.dirname(self.current_directory)
        self.fill_store()
        sensitive = True
        if self.current_directory == "/": sensitive = False
        self.upButton.set_sensitive(sensitive)


PyApp()
gtk.main()
```

This example shows icons of the currently selected directory. It has a toolbar and two buttons. Up button and home button. We use them to navigate through the file system.

```
self.current_directory = '/'
```

The **current_directory** is the directory, that is displayed by the **IconView** widget.

```
def create_store(self):
    store = gtk.ListStore(str, gtk.gdk.Pixbuf, bool)
    store.set_sort_column_id(COL_PATH, gtk.SORT_ASCENDING)
```

```
    return store
```

The **create_store()** method creates a **ListStore**. It is a data model used in **IconView** widget. It takes three parameters. The directory name, the pixbuf image of the icon and a bool variable, indicating, whether we have a directory or a file.

```
if not fl[0] == '.':
    if os.path.isdir(os.path.join(self.current_directory, fl)):
        self.store.append([fl, self.dirIcon, True])
    else:
        self.store.append([fl, self.fileIcon, False])
```

In the **fill_store()** method, we fill the list store with data. Here, we find out all directories in the current path. We exclude the invisible directories, which begin with '.'.

```
def on_home_clicked(self, widget):
    self.current_directory =
os.path.realpath(os.path.expanduser('~'))
    self.fill_store()
    self.upButton.set_sensitive(True)
```

If we click on the home button, the home directory becomes a current directory. We refill the list store. And make the up button active.

In the **on_item_activated()** method, we react to an event, which is generated, when we click on a icon from the icon view widget.

```
model = widget.get_model()
path = model[item][COL_PATH]
isDir = model[item][COL_IS_DIRECTORY]

if not isDir:
    return
```

We get the path of the activated item. And we determine, if it is a directory or a file. If it is a file, we return.

```
self.current_directory = self.current_directory + os.path.sep + path
self.fill_store()
self.upButton.set_sensitive(True)
```

In case it is a directory, we replace the root with the current path, refill the store and make the up button sensitive.

```
def on_up_clicked(self, widget):
    self.current_directory = os.path.dirname(self.current_directory)
    self.fill_store()
    sensitive = True
    if self.current_directory == "/": sensitive = False
    self.upButton.set_sensitive(sensitive)
```

If we click on the up button, we replace the current directory with it's parent directory. Refill the list store. And the up button is activated, if we are below the root (/) directory of the file system.

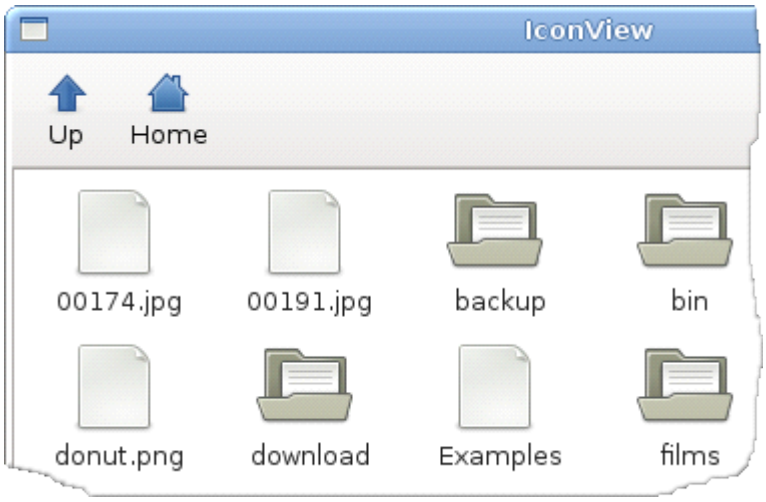


Figure: IconView

ListView

In the following example, we use the **TreeView** widget to show a list view. Again the **ListStore** is used to store data.

listview.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows a TreeView widget
# in a list view mode
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
```

```
actresses = [('jessica alba', 'pomona', '1981'), ('sigourney weaver', 'new
york', '1949'),
    ('angelina jolie', 'los angeles', '1975'), ('natalie portman',
'jerusalem', '1981'),
    ('rachel weiss', 'london', '1971'), ('scarlett johansson', 'new york',
'1984' )]


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(350, 250)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)
        self.set_title("ListView")

        vbox = gtk.VBox(False, 8)

        sw = gtk.ScrolledWindow()
        sw.set_shadow_type(gtk.SHADOW_ETCHED_IN)
        sw.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)

        vbox.pack_start(sw, True, True, 0)

        store = self.create_model()

        treeView = gtk.TreeView(store)
        treeView.connect("row-activated", self.on_activated)
        treeView.set_rules_hint(True)
        sw.add(treeView)

        self.create_columns(treeView)
        self.statusbar = gtk.Statusbar()

        vbox.pack_start(self.statusbar, False, False, 0)

        self.add(vbox)
        self.show_all()


    def create_model(self):
        store = gtk.ListStore(str, str, str)

        for act in actresses:
            store.append([act[0], act[1], act[2]])

        return store


    def create_columns(self, treeView):

        rendererText = gtk.CellRendererText()
        column = gtk.TreeViewColumn("Name", rendererText, text=0)
        column.set_sort_column_id(0)
        treeView.append_column(column)
```

```
        rendererText = gtk.CellRendererText()
        column = gtk.TreeViewColumn("Place", rendererText, text=1)
        column.set_sort_column_id(1)
        treeView.append_column(column)

        rendererText = gtk.CellRendererText()
        column = gtk.TreeViewColumn("Year", rendererText, text=2)
        column.set_sort_column_id(2)
        treeView.append_column(column)


    def on_activated(self, widget, row, col):

        model = widget.get_model()
        text = model[row][0] + ", " + model[row][1] + ", " + model[row][2]
        self.statusbar.push(0, text)



PyApp()
gtk.main()
```

In our example, we show a list of six actresses in the **TreeView** widget. Each of the rows shows the name, the place of born and the year of born for each of them.

```
def create_model(self):
    store = gtk.ListStore(str, str, str)

    for act in actresses:
        store.append([act[0], act[1], act[2]])

    return store
```

In the **create_model()** method, we create the list store. The list store has three parameters. The name of the actress, the place of born and year of born. This is the data model of our **TreeView** widget.

```
treeView = gtk.TreeView(store)
treeView.connect("row-activated", self.on_activated)
treeView.set_rules_hint(True)
```

Here we create the **TreeView** widget, taking the list store as a parameter. **set_rules_hint()** method changes the background color of the every second row in the **TreeView** widget.

```
rendererText = gtk.CellRendererText()

column = gtk.TreeViewColumn("Name", rendererText, text=0)
```

```
column.set_sort_column_id(0)
treeView.append_column(column)
```

In the **create_columns()** method, we add three columns to our **TreeView** widget. The above code creates a column displaying names of the actresses. The **CellRendererText** retrieves its text from the first column of the tree model. (text=0)

```
def on_activated(self, widget, row, col):

    model = widget.get_model()
    text = model[row][0] + ", " + model[row][1] + ", " + model[row][2]
    self.statusbar.push(0, text)
```

If we double click on an item, we display the whole row in the statusbar.

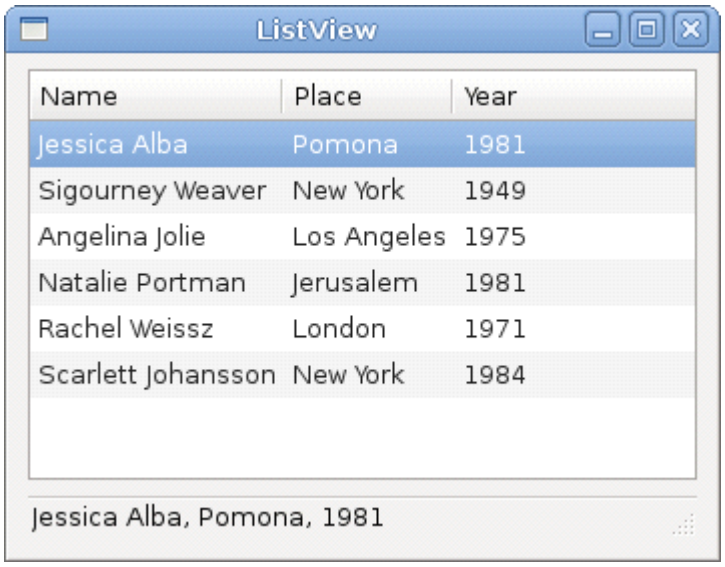


Figure: ListView

Tree

In the last example of this chapter, we use the **TreeView** widget to show a hierarchical tree of data.

tree.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
```

```
# This example shows a TreeView widget
# in a tree view mode
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(400, 300)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)
        self.set_title("Tree")

        tree = gtk.TreeView()

        languages = gtk.TreeViewColumn()
        languages.set_title("Programming languages")

        cell = gtk.CellRendererText()
        languages.pack_start(cell, True)
        languages.add_attribute(cell, "text", 0)

        treestore = gtk.TreeStore(str)

        it = treestore.append(None, ["Scripting languages"])
        treestore.append(it, ["Python"])
        treestore.append(it, ["PHP"])
        treestore.append(it, ["Perl"])
        treestore.append(it, ["Ruby"])

        it = treestore.append(None, ["Compiling languages"])
        treestore.append(it, ["C#"])
        treestore.append(it, ["C++"])
        treestore.append(it, ["C"])
        treestore.append(it, ["Java"])

        tree.append_column(languages)
        tree.set_model(treestore)

        self.add(tree)
        self.show_all()


PyApp()
gtk.main()
```

This time we use the **TreeView** widget to show hierarchical data.

```
 tree = gtk.TreeView()
```

TreeView widget is created.

```
languages = gtk.TreeViewColumn()
languages.set_title("Programming languages")
```

It has one column named "Programming languages".

```
cell = gtk.CellRendererText()
languages.pack_start(cell, True)
languages.add_attribute(cell, "text", 0)
```

We show textual data in the TreeView widget.

```
treestore = gtk.TreeStore(str)
```

To store the data, we use the TreeStore object.

```
it = treestore.append(None, ["Scripting languages"])
treestore.append(it, ["Python"])
treestore.append(it, ["PHP"])
```

We append data to the tree. The TreeIter object is used for accessing data in a row.

```
tree.append_column(languages)
```

A column is appended to the tree.

```
tree.set_model(treestore)
```

Finally, we set a data model for the tree widget.

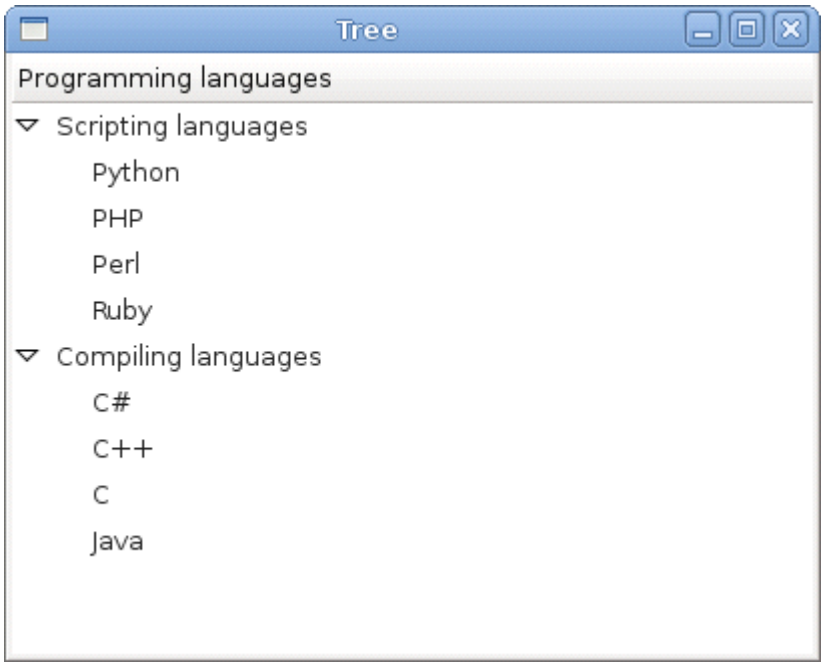


Figure: Tree

In this chapter of the PyGTK programming tutorial, we were talking about advanced PyGTK widgets.

Dialogs in PyGTK

In this part of the PyGTK programming tutorial, we will introduce dialogs.

Dialog windows or dialogs are an indispensable part of most modern GUI applications. A dialog is defined as a conversation between two or more persons. In a computer application a dialog is a window which is used to "talk" to the application. And vice versa. A dialog is used to input data, modify data, change the application settings etc. Dialogs are important means of communication between a user and a computer program.

Message dialogs

Message dialogs are convenient dialogs that provide messages to the user of the application. The message consists of textual and image data.

messages.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows message
# dialogs
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(250, 100)
        self.set_position(gtk.WIN_POS_CENTER)
```

```
        self.connect("destroy", gtk.main_quit)
        self.set_title("Message dialogs")


        table = gtk.Table(2, 2, True);

        info = gtk.Button("Information")
        warn = gtk.Button("Warning")
        ques = gtk.Button("Question")
        erro = gtk.Button("Error")


        info.connect("clicked", self.on_info)
        warn.connect("clicked", self.on_warn)
        ques.connect("clicked", self.on_ques)
        erro.connect("clicked", self.on_erro)

        table.attach(info, 0, 1, 0, 1)
        table.attach(warn, 1, 2, 0, 1)
        table.attach(ques, 0, 1, 1, 2)
        table.attach(erro, 1, 2, 1, 2)


        self.add(table)
        self.show_all()

    def on_info(self, widget):
        md = gtk.MessageDialog(self,
            gtk.DIALOG_DESTROY_WITH_PARENT, gtk.MESSAGE_INFO,
            gtk.BUTTONS_CLOSE, "Download completed")
        md.run()
        md.destroy()


    def on_erro(self, widget):
        md = gtk.MessageDialog(self,
            gtk.DIALOG_DESTROY_WITH_PARENT, gtk.MESSAGE_ERROR,
            gtk.BUTTONS_CLOSE, "Error loading file")
        md.run()
        md.destroy()



    def on_ques(self, widget):
        md = gtk.MessageDialog(self,
            gtk.DIALOG_DESTROY_WITH_PARENT, gtk.MESSAGE_QUESTION,
            gtk.BUTTONS_CLOSE, "Are you sure to quit?")
        md.run()
        md.destroy()


    def on_warn(self, widget):
        md = gtk.MessageDialog(self,
            gtk.DIALOG_DESTROY_WITH_PARENT, gtk.MESSAGE_WARNING,
            gtk.BUTTONS_CLOSE, "Unallowed operation")
        md.run()
        md.destroy()
```

```
PyApp()
gtk.main()
```

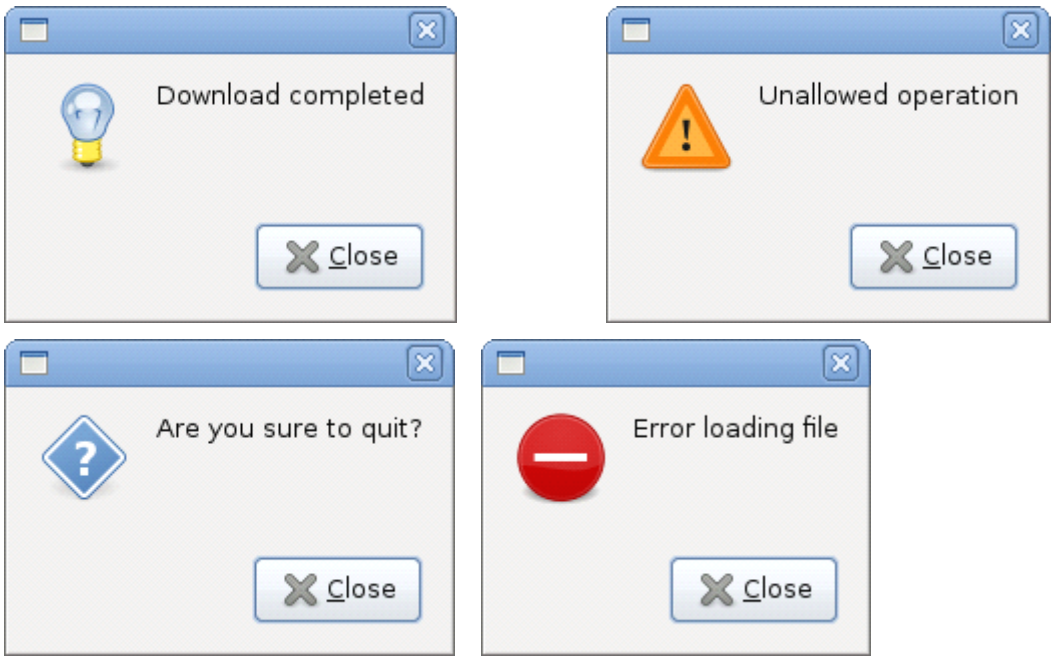
In our example, we will show four kinds of message dialogs. Information, Warning, Question and Error message dialogs.

```
info = gtk.Button("Information")
warn = gtk.Button("Warning")
ques = gtk.Button("Question")
erro = gtk.Button("Error")
```

We have four buttons. Each of these buttons will show a different kind of message dialog.

```
md = gtk.MessageDialog(self,
    gtk.DIALOG_DESTROY_WITH_PARENT, gtk.MESSAGE_INFO,
        gtk.BUTTONS_CLOSE, "Download completed")
md.run()
md.destroy()
```

If we click on the info button, the Information dialog is displayed. The **MESSAGE_INFO** specifies the type of the dialog. The **BUTTONS_CLOSE** specifies the button to be displayed in the dialog. The last parameter is the message dislayed. The dialog is displayed with the **run()** method. The programmer must also call either the **destroy()** or the **hide()** method.



AboutDialog

The AboutDialog displays information about the application. AboutDialog can display a logo, the name of the application, version, copyright, website or licence information. It is also possible to give credits to the authors, documenters, translators and artists.

aboutdialog.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example demonstrates the
# AboutDialog dialog
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()
        self.set_size_request(300, 150)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)
        self.set_title("About battery")


        button = gtk.Button("About")
        button.set_size_request(80, 30)
        button.connect("clicked", self.on_clicked)

        fix = gtk.Fixed()
        fix.put(button, 20, 20)

        self.add(fix)
        self.show_all()

    def on_clicked(self, widget):
        about = gtk.AboutDialog()
        about.set_program_name("Battery")
        about.set_version("0.1")
        about.set_copyright("(c) Jan Bodnar")
        about.set_comments("Battery is a simple tool for battery checking")
        about.set_website("http://www.zetcode.com")
        about.set_logo(gtk.gdk.pixbuf_new_from_file("battery.png"))
        about.run()
        about.destroy()
```

```
PyApp()
gtk.main()
```

The code example uses a **AboutDialog** with some of it's features.

```
 about = gtk.AboutDialog()
```

We create an **AboutDialog**.

```
 about = gtk.AboutDialog()
 about.set_program_name("Battery")
 about.set_version("0.1")
 about.set_copyright("(c) Jan Bodnar")
```

We specify the name, version and the copyright.

```
 about.set_logo(gtk.gdk.pixbuf_new_from_file("battery.png"))
```

This line creates a logo.

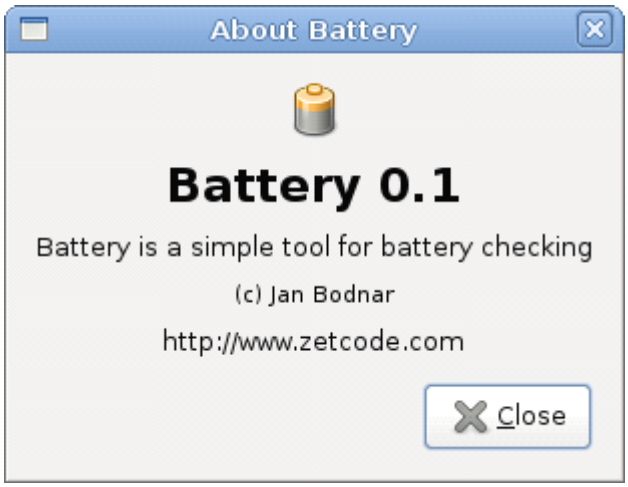


Figure: AboutDialog

FontSelectionDialog

The **FontSelectionDialog** is a dialog for selecting fonts. It is typically used in applications, that do some text editing or formatting.

fontdialog.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
```

```
#
# This example works with the
# FontSelection dialog
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import pango


class PyApp(gtk.Window):
    def __init__(self):
        gtk.Window.__init__(self)
        self.set_size_request(300, 150)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)
        self.set_title("Font Selection Dialog")


        self.label = gtk.Label("The only victory over love is flight.")
        button = gtk.Button("Select font")
        button.connect("clicked", self.on_clicked)

        fix = gtk.Fixed()
        fix.put(button, 100, 30)
        fix.put(self.label, 30, 90)
        self.add(fix)

        self.show_all()

    def on_clicked(self, widget):
        fdia = gtk.FontSelectionDialog("Select font name")
        response = fdia.run()

        if response == gtk.RESPONSE_OK:
            font_desc = pango.FontDescription(fdia.get_font_name())
            if font_desc:
                self.label.modify_font(font_desc)

        fdia.destroy()


PyApp()
gtk.main()
```

In the code example, we have a button and a label. We show the FontSelectionDialog by clicking on the button.

```
 fdia = gtk.FontSelectionDialog("Select font name")
```

We create the FontSelectionDialog.

```
if response == gtk.RESPONSE_OK:
    font_desc = pango.FontDescription(fdia.get_font_name())
    if font_desc:
        self.label.modify_font(font_desc)
```

If we click on the OK button, the font of the label widget changes to the one, that we selected in the dialog.

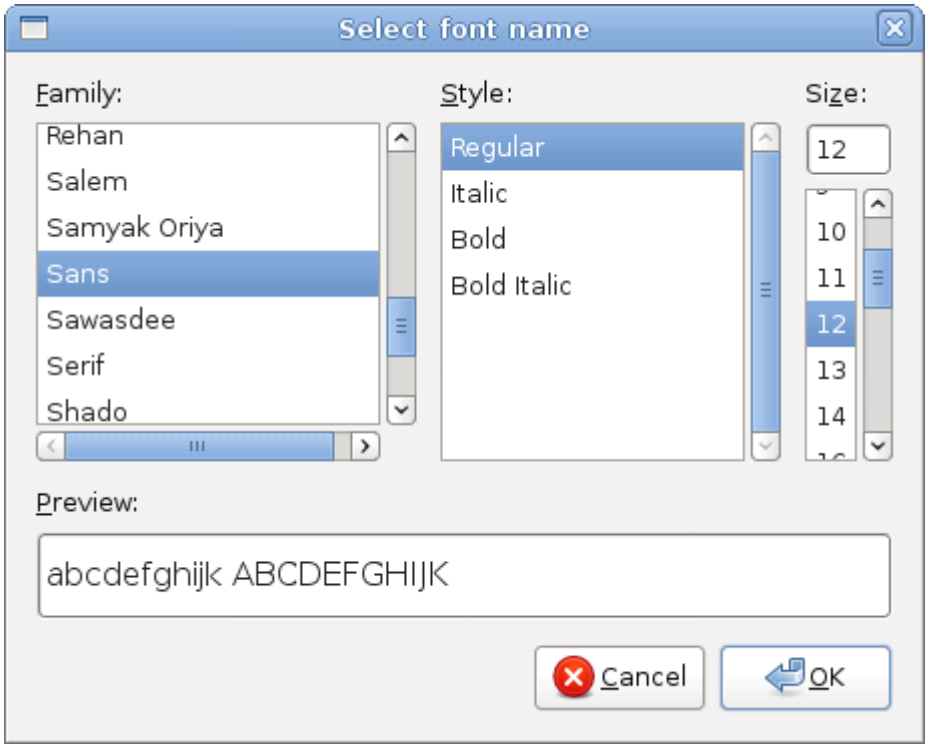


Figure: FontSelectionDialog

ColorSelectionDialog

ColorSelectionDialog is a dialog for selecting a color.

colordialog.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example works with the
# ColorSelection dialog
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009
```

```
import gtk


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(300, 150)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)
        self.set_title("Color Selection Dialog")


        self.label = gtk.Label("The only victory over love is flight.")
        button = gtk.Button("Select color")
        button.connect("clicked", self.on_clicked)

        fix = gtk.Fixed()
        fix.put(button, 100, 30)
        fix.put(self.label, 30, 90)
        self.add(fix)

        self.show_all()

    def on_clicked(self, widget):
        cdia = gtk.ColorSelectionDialog("Select color")
        response = cdia.run()

        if response == gtk.RESPONSE_OK:
            colorsel = cdia.colorsel
            color = colorsel.get_current_color()
            self.label.modify_fg(gtk.STATE_NORMAL, color)

        cdia.destroy()


PyApp()
gtk.main()
```

The example is very similar to the previous one. This time we change the color of the label.

```
 cdia = gtk.ColorSelectionDialog("Select color")
```

We create the ColorSelectionDialog.

```
 if response == gtk.RESPONSE_OK:
     colorsel = cdia.colorsel
     color = colorsel.get_current_color()
     self.label.modify_fg(gtk.STATE_NORMAL, color)
```

If the user pressed OK, we get the color and modify the label's color.

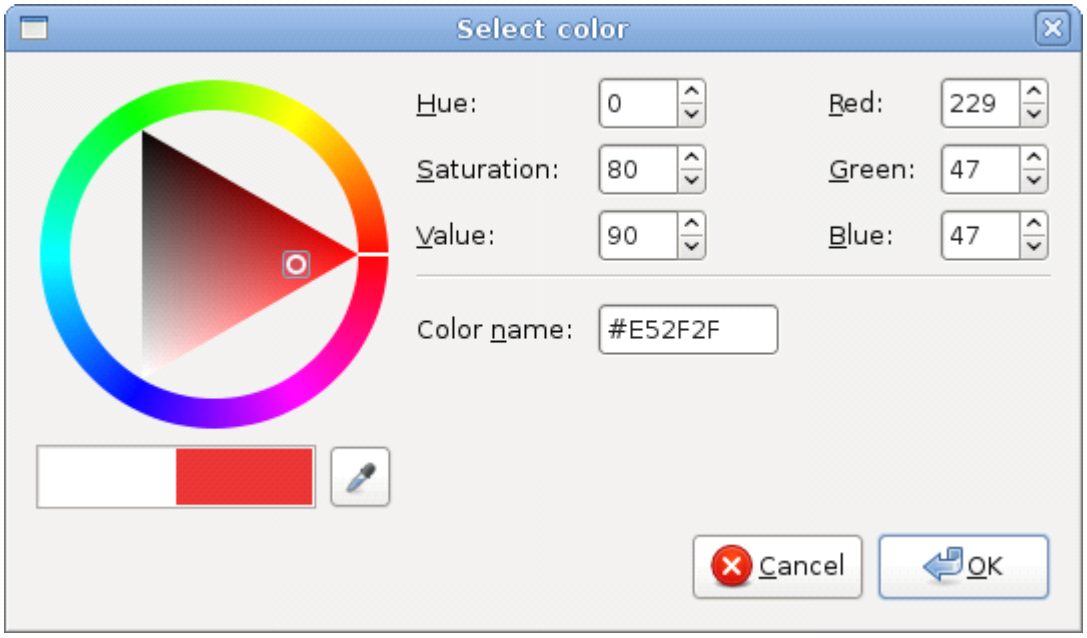


Figure: ColorSelectionDialog

In this part of the PyGTK programming tutorial, we worked with PyGTK built-in dialogs.

Pango

In this part of the PyGTK programming tutorial, we will explore the Pango library.

Pango is a free and open source computing library for rendering internationalized texts in high quality. Different font backends can be used, allowing cross-platform support. (wikipedia)

Pango provides advanced font and text handling that is used for **Gdk** and **Gtk**.

Simple example

In our first example, we show, how to change font for our label widget.

quotes.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows how to modify
# the font of a label
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import pango

quotes = """Excess of joy is harder to bear than any amount of sorrow.
The more one judges, the less one loves.
There is no such thing as a great talent without great will power.
"""


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.connect("destroy", gtk.main_quit)
        self.set_title("Quotes")

        label = gtk.Label(quotes)
        gtk.gdk.beep()

        fontdesc = pango.FontDescription("Purisa 10")
        label.modify_font(fontdesc)

        fix = gtk.Fixed()

        fix.put(label, 5, 5)

        self.add(fix)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()

PyApp()
gtk.main()
```

In the above code example, we have a label widget with three quotations. We change it's font to Purisa 10.

```
 quotes = """Excess of joy is harder to bear than any amount of sorrow.
 The more one judges, the less one loves.
 There is no such thing as a great talent without great will power.
"""
```

This is the text to show in the label.

```
fontdesc = pango.FontDescription("Purisa 10")
```

The **FontDescription** is used to specify the characteristics of a font.

```
label.modify_font(fontdesc)
```

We change the font of the label widget to Purisa 10.

Figure: Quotations

System fonts

The next code example shows all available fonts in a **TreeView** widget.

systemfonts.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example lists all available
# fonts on a system in a TreeView widget
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import pango


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_size_request(350, 250)
        self.set_border_width(8)
        self.connect("destroy", gtk.main_quit)
        self.set_title("System fonts")

        sw = gtk.ScrolledWindow()
        sw.set_shadow_type(gtk.SHADOW_ETCHED_IN)
        sw.set_policy(gtk.POLICY_AUTOMATIC, gtk.POLICY_AUTOMATIC)
```

```
        context = self.create_pango_context()
        self.fam = context.list_families()

        store = self.create_model()

        treeView = gtk.TreeView(store)
        treeView.set_rules_hint(True)
        sw.add(treeView)

        self.create_column(treeView)

        self.add(sw)

        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()


    def create_column(self, treeView):
        rendererText = gtk.CellRendererText()
        column = gtk.TreeViewColumn("FontName", rendererText, text=0)
        column.set_sort_column_id(0)
        treeView.append_column(column)

    def create_model(self):
        store = gtk.ListStore(str)

        for ff in self.fam:
            store.append([ff.get_name()])

        return store


PyApp()
gtk.main()
```

The code example shows all available fonts on a system.

```
context = self.create_pango_context()
```

This code line creates a pango context object. It contains global information about the rendering process of text.

```
self.fam = context.list_families()
```

From the context object, we retrieve all available font families.

```
for ff in self.fam:
    store.append([ff.get_name()])
```

During the model creation of the **TreeView** widget, we get all font names from the array of font families and put them into the list store.

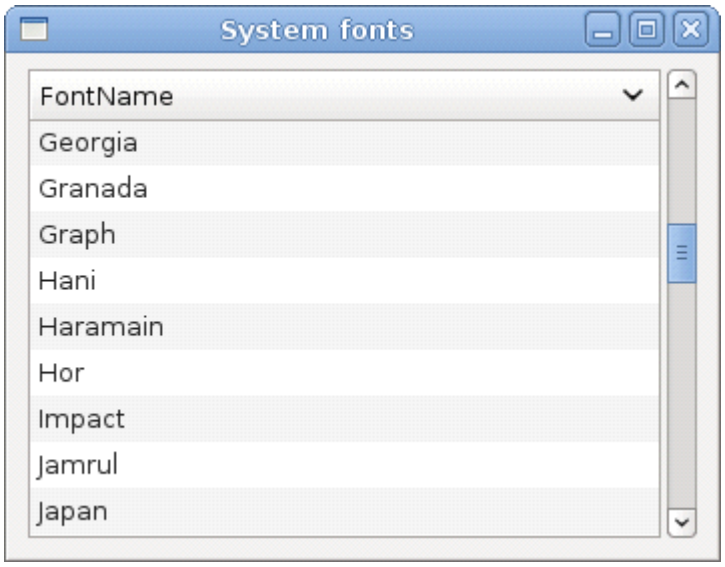


Figure: System fonts

Unicode

Pango is used to work with internationalized text.

unicode.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# ZetCode PyGTK tutorial
#
# This example displays text
# in azbuka
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import pango


obj = unicode(u'''Фёдор Михайлович Достоевский родился 30 октября (11
ноября)
1821 года в Москве. Был вторым из 7 детей. Отец, Михаил Андреевич,
работал вгоспитале для бедных. Мать, Мария Фёдоровна
(в девичестве Нечаева), происходила из купеческого рода.''')


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()
```

```
        self.connect("destroy", gtk.main_quit)
        self.set_title("Unicode")

        label = gtk.Label(obj.encode('utf-8'))

        fontdesc = pango.FontDescription("Purisa 10")
        label.modify_font(fontdesc)

        fix = gtk.Fixed()

        fix.put(label, 5, 5)

        self.add(fix)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()

PyApp()
gtk.main()
```

We show some text in azbuka.

```
 # -*- coding: utf-8 -*-
```

In order to work directly with internationalized text in the source code, we must provide this magic comment. Note, that it must be on the first or the second line.

```
 obj = unicode(u'''Фёдор Михайлович
Достоевский родился 30 октября (11
ноября)
1821 года в Москве. Был вторым из 7 детей.
Отец, Михаил Андреевич,
работал вгоспитале для бедных. Мать,
Мария Фёдоровна
(в девичестве Нечаева), происходила из
купеческого рода.''')
```

This is text in azbuka.

```
 Label label = new Label(text);
```

We put encoded text into the label.

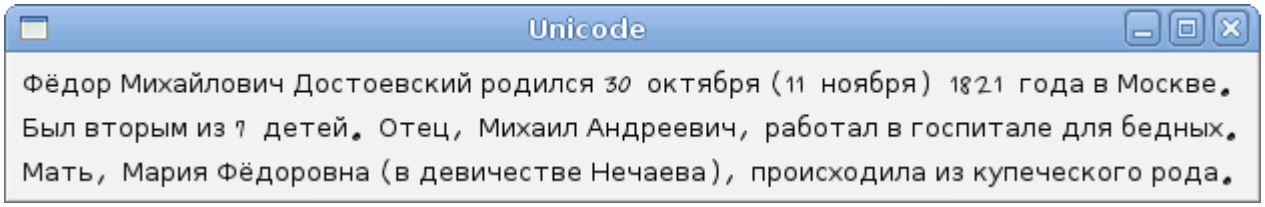


Figure: Unicode

Attributes

Pango attribute is an attribute that applies to a section of text.

attributes.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# In this program we work with
# pango attributes
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import pango

text = "Valour fate kinship darkness"


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.connect("destroy", gtk.main_quit)
        self.set_title("Attributes")

        label = gtk.Label(text)

        attr = pango.AttrList()

        fg_color = pango.AttrForeground(65535, 0, 0, 0, 6)
        underline = pango.AttrUnderline(pango.UNDERLINE_DOUBLE, 7, 11)
        bg_color = pango.AttrBackground(40000, 40000, 40000, 12, 19)
        strike = pango.AttrStrikethrough(True, 20, 29)
        size = pango.AttrSize(30000, 0, -1)

        attr.insert(fg_color)
        attr.insert(underline)
        attr.insert(bg_color)
        attr.insert(size)
        attr.insert(strike)

        label.set_attributes(attr)

        fix = gtk.Fixed()

        fix.put(label, 5, 5)
```

```
        self.add(fix)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()

PyApp()
gtk.main()
```

In the code example we show four different attributes applied on the text.

```
attr = pango.AttrList()
```

Pango attribute list is an object for holding attributes.

```
fg_color = pango.AttrForeground(65535, 0, 0, 0, 6)
```

Here we create an attribute that will render text in red color. The first three parameters are the R, G, B values of a color. The last two parameters are the start and end indexes of the text, to which we apply this attribute.

```
label.set_attributes(attr)
```

We set the list of attributes for the label.

Figure: Pango attributes

In this chapter of the PyGTK programming library, we worked with pango library.

Pango

In this part of the PyGTK programming tutorial, we will continue exploring the Pango library.

Animated text

The following example shows animated text on window.

animation.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows animated text
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import glib
import pango
import math


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.connect("destroy", gtk.main_quit)
        glib.timeout_add(160, self.on_timer)

        self.count = 1

        self.set_border_width(10)
        self.set_title("ZetCode")

        self.label = gtk.Label("ZetCode")

        fontdesc = pango.FontDescription("Serif Bold 30")
        self.label.modify_font(fontdesc)

        vbox = gtk.VBox(False, 0)
        vbox.add(self.label)

        self.add(vbox)
        self.set_size_request(300, 250)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()

    def on_timer(self):
        attr = pango.AttrList()
        self.count = self.count + 1

        for i in range(7):
            r = pango.AttrRise(int(math.sin(self.count+i)*20)*pango.SCALE,
i, i+1)
            attr.insert(r)
```

```
        self.label.set_attributes(attr)
        return True


PyApp()
gtk.main()
```

In the above code example, we have a text in a label widget. By continuously changing its pango attributes, the text is being animated.

```
 self.label = gtk.Label("ZetCode")

 fontdesc = pango.FontDescription("Serif Bold 30")
 self.label.modify_font(fontdesc)
```

We create a label widget and modify its font. We choose a bit larger text for better visibility.

```
 vbox = gtk.VBox(False, 0)
 vbox.add(self.label)
```

We put the label into the vertical box. This centers the label on the window.

The animation is performed inside the **on_timer()** method.

```
 for i in range(7):
     r = pango.AttrRise(int(math.sin(self.count+i)*20)*pango.SCALE, i,
i+1)
     attr.insert(r)
```

We have seven characters in our text. We periodically change the pango **AttrRise** attribute for each character. The rise is based on the trigonometric sine function. The text movement follows the sine function graphed on the cartesian graph.

Also notice the **pango.SCALE** constant. The pango library has its own units. They differ from what is used by the widgets to draw graphics or text. We must multiply our numbers by this constant.

Figure: Animated text

Using markup language

We can change the attributes of the text using the built-in markup language.

markup.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example uses markup language
# to change attributes of the text
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import pango

quote = "<span foreground='blue' size='19000'>The only victory over love
is flight</span>"


class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Markup")
        self.set_border_width(5)
        self.connect("destroy", gtk.main_quit)
```

```
        label = gtk.Label()
        label.set_markup(quote)

        vbox = gtk.VBox(False, 0)
        vbox.add(label)

        self.add(vbox)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()

PyApp()
gtk.main()
```

In the code example, we have a label. We change the it's text attributes with the markup language.

```
 quote = "<span foreground='blue' size='19000'>The only victory over love is flight</span>"
```

This is the text with the markup language.

```
 label = gtk.Label()
 label.set_markup(quote)
```

We create a label widget and set a markup text for it.

Figure: Using markup

Pango layout

Pango layout is an object representing a paragraph of text with attributes.

layout.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This example shows pango Layout
# in action
```

```
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import pango


lyrics = """Meet you downstairs in the bar and heard
your rolled up sleeves and your skull t-shirt
You say why did you do it with him today?
and sniff me out like I was Tanqueray

cause you're my fella, my guy
hand me your stella and fly
by the time I'm out the door
you tear men down like Roger Moore

I cheated myself
like I knew I would
I told ya, I was trouble
you know that I'm no good"""

class Area(gtk.DrawingArea):
    def __init__(self):
        super(Area, self).__init__()
        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(16400, 16400,
16440))
        self.connect("expose_event", self.expose)

    def expose(self, widget, event):

        gc = self.get_style().fg_gc[gtk.STATE_NORMAL]
        font_desc = pango.FontDescription('Sans 10')

        layout = self.create_pango_layout(lyrics)
        width, height = self.get_size_request()

        attr = pango.AttrList()

        fg_color = pango.AttrForeground(60535, 60535, 60535, 0, -1)
        attr.insert(fg_color)
```

```
        layout.set_width(pango.SCALE * self.allocation.width)
        layout.set_spacing(pango.SCALE * 3)
        layout.set_alignment(pango.ALIGN_CENTER)
        layout.set_font_description(font_desc)
        layout.set_attributes(attr)

        self.window.draw_layout(gc, 0, 5, layout)



class PyApp(gtk.Window):
    def __init__(self):
        super(PyApp, self).__init__()

        self.connect("destroy", gtk.main_quit)
        self.set_title("You know I'm no Good")

        self.add(Area())
        self.set_size_request(300, 300)
        self.set_position(gtk.WIN_POS_CENTER)
        self.show_all()


PyApp()
gtk.main()
```

In the previous examples, we were modifying text in existing widgets. Now we are going to draw the text using the pango layout on the **DrawingArea** widget. We will be drawing using the **Gdk** drawing tools.

```
 gc = self.get_style().fg_gc[gtk.STATE_NORMAL]
```

We get the graphics contex of the drawing area widget.

```
 layout = self.create_pango_layout(lyrics)
```

Here create the pango layout object.

```
 layout.set_width(pango.SCALE * self.allocation.width)
 layout.set_spacing(pango.SCALE * 3)
 layout.set_alignment(pango.ALIGN_CENTER)
 layout.set_font_description(font_desc)
 layout.set_attributes(attr)
```

We modify layout's width, spacing, alignment, font and set text attributes.

```
self.window.draw_layout(gc, 0, 5, layout)
```

The layout is being drawn on the window.

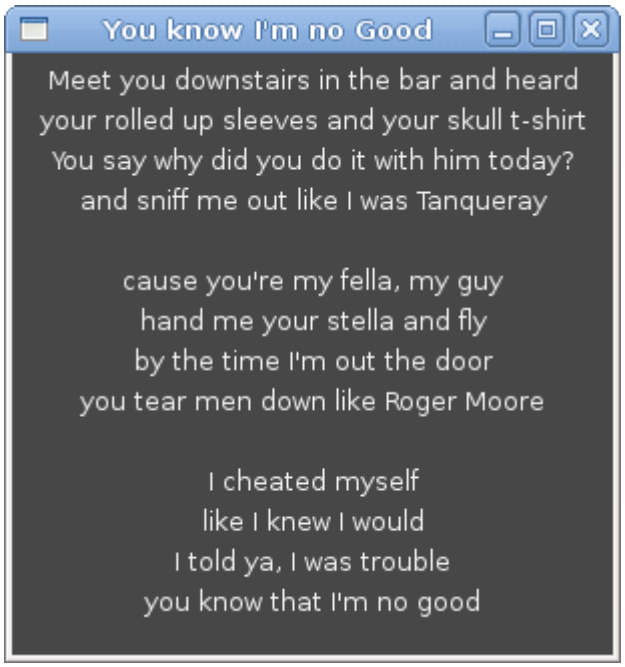


Figure: Layout

In this chapter of the PyGTK programming library, we further worked with pango library.

Drawing with Cairo

In this part of the PyGTK programming tutorial, we will do some drawing with the Cairo library.

Cairo is a library for creating 2D vector graphics. We can use it to draw our own widgets, charts or various effects or animations.

Simple drawing

The stroke operation draws the outlines of shapes and the fill operation fills the insides of shapes. Next we will demonstrate these two operations.

simpledrawing.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This code example draws a circle
# using the cairo library
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import math

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Simple drawing")
        self.resize(230, 150)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()
```

```
        cr.set_line_width(9)
        cr.set_source_rgb(0.7, 0.2, 0.0)

        w = self.allocation.width
        h = self.allocation.height

        cr.translate(w/2, h/2)
        cr.arc(0, 0, 50, 0, 2*math.pi)
        cr.stroke_preserve()

        cr.set_source_rgb(0.3, 0.4, 0.6)
        cr.fill()


PyApp()
gtk.main()
```

In our example, we will draw a circle and will it with a solid color.

```
 darea = gtk.DrawingArea()
```

We will be doing our drawing operations on the **DrawingArea** widget.

```
 darea.connect("expose-event", self.expose)
```

We do all drawing in a method, that is a handler for the **expose-event** signal.

```
 cr = widget.window.cairo_create()
```

We create the cairo context object from the **gdk.Window** of the drawing area. The context is an object that is used to draw on all Drawable objects.

```
 cr.set_line_width(9)
```

We set the width of the line to 9 pixels.

```
 cr.set_source_rgb(0.7, 0.2, 0.0)
```

We set the color to dark red.

```
 w = self.allocation.width
 h = self.allocation.height
```

```
cr.translate(w/2, h/2)
```

We get the width and height of the drawing area. We move the origin into the middle of the window.

```
cr.arc(0, 0, 50, 0, 2*math.pi)
cr.stroke_preserve()
```

We draw the outside shape of a circle. In red color. The **stroke_preserve()** strokes the current path according to the current line width, line join, line cap, and dash settings. Unlike the **stroke()**, it preserves the path within the cairo context.

```
cr.set_source_rgb(0.3, 0.4, 0.6)
cr.fill()
```

This fills the interior of the circle with some blue color.

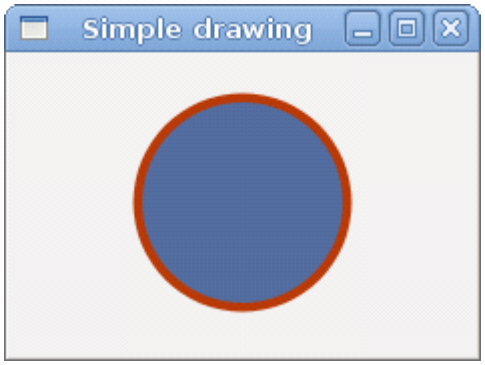


Figure: Simple drawing

Basic shapes

The next example draws some basic shapes onto the window.

basicshapes.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This code example draws basic shapes
# with the cairo library
#
```

```
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import math

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Basic shapes")
        self.set_size_request(390, 240)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()
        cr.set_source_rgb(0.6, 0.6, 0.6)

        cr.rectangle(20, 20, 120, 80)
        cr.rectangle(180, 20, 80, 80)
        cr.fill()

        cr.arc(330, 60, 40, 0, 2*math.pi)
        cr.fill()

        cr.arc(90, 160, 40, math.pi/4, math.pi)
        cr.fill()

        cr.translate(220, 180)
        cr.scale(1, 0.7)
        cr.arc(0, 0, 50, 0, 2*math.pi)
        cr.fill()
```

```
PyApp()
gtk.main()
```

In this example, we will create a rectangle, a square, a circle, an arc and an ellipse.

```
 cr.rectangle(20, 20, 120, 80)
 cr.rectangle(180, 20, 80, 80)
 cr.fill()
```

These lines draw a rectangle and a square.

```
 cr.arc(330, 60, 40, 0, 2*math.pi)
 cr.fill()
```

Here the **arc()** method draws a full circle.

```
 cr.scale(1, 0.7)
 cr.arc(0, 0, 50, 0, 2*math.pi)
 cr.fill()
```

If we want to draw an oval, we do some scaling first. Here the **scale()** method shrinks the y axis.

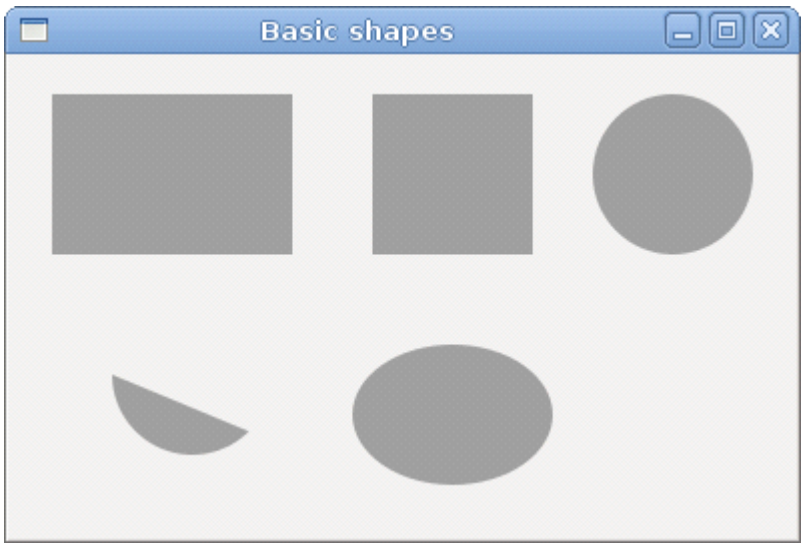


Figure: Basic shapes

Colors

A color is an object representing a combination of Red, Green, and Blue (RGB) intensity values. Cairo valid RGB values are in the range 0 to 1.

colors.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program shows how to work
# with colors in cairo
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Colors")
        self.resize(360, 100)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()

        cr.set_source_rgb(0.2, 0.23, 0.9)
```

```
        cr.rectangle(10, 15, 90, 60)
        cr.fill()

        cr.set_source_rgb(0.9, 0.1, 0.1)
        cr.rectangle(130, 15, 90, 60)
        cr.fill()

        cr.set_source_rgb(0.4, 0.9, 0.4)
        cr.rectangle(250, 15, 90, 60)
        cr.fill()

PyApp()
gtk.main()
```

We draw three rectangles in three different colors.

```
 cr.set_source_rgb(0.2, 0.23, 0.9)
```

The **set_source_rgb()** method sets a color for the cairo context. The three parameters of the method are the color intensity values.

```
 cr.rectangle(10, 15, 90, 60)
 cr.fill()
```

We create a rectangle shape and fill it with the previously specified color.

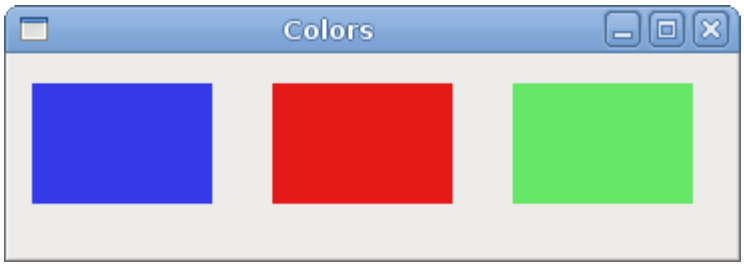


Figure: Colors

Transparent rectangles

Transparency is the quality of being able to see through a material. The easiest way to understand transparency is to imagine a piece of glass or water. Technically, the rays of light can go through the glass and this way we can see objects behind the glass.

In computer graphics, we can achieve transparency effects using alpha compositing. Alpha compositing is the process of combining an image with a background to create the appearance of partial transparency. The composition process uses an alpha channel. (wikipedia.org, answers.com)

transparentrectangles.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program shows transparent
# rectangles using cairo
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Transparent rectangles")
        self.resize(590, 90)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()

        for i in range(1, 11):
            cr.set_source_rgba(0, 0, 1, i*0.1)
```

```
            cr.rectangle(50*i, 20, 40, 40)
            cr.fill()


PyApp()
gtk.main()
```

In the example we will draw ten rectangles with different levels of transparency.

```
 cr.set_source_rgba(0, 0, 1, i*0.1)
```

The last parameter of the **set_source_rgba()** method is the alpha transparency.

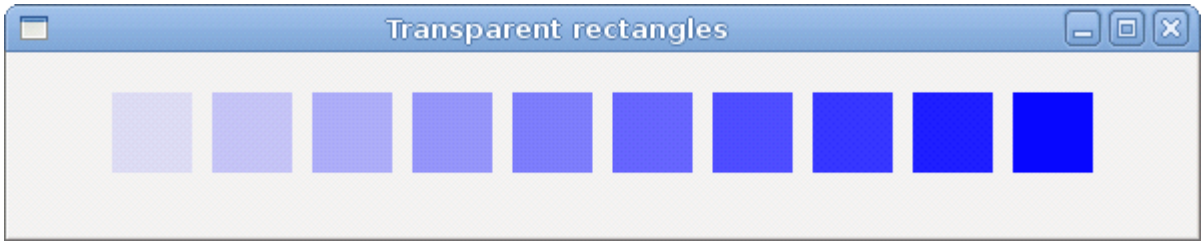


Figure: Transparent rectangles

Soulmate

In the next example, we draw some text on the window.

soulmate.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program draws text
# using cairo
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
```

```
import cairo

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Soulmate")
        self.set_size_request(370, 240)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()

        cr.set_source_rgb(0.1, 0.1, 0.1)

        cr.select_font_face("Purisa", cairo.FONT_SLANT_NORMAL,
            cairo.FONT_WEIGHT_NORMAL)
        cr.set_font_size(13)

        cr.move_to(20, 30)
        cr.show_text("Most relationships seem so transitory")
        cr.move_to(20, 60)
        cr.show_text("They're all good but not the permanent one")
        cr.move_to(20, 120)
        cr.show_text("Who doesn't long for someone to hold")
        cr.move_to(20, 150)
        cr.show_text("Who knows how to love without being told")
        cr.move_to(20, 180)
        cr.show_text("Somebody tell me why I'm on my own")
        cr.move_to(20, 210)
        cr.show_text("If there's a soulmate for everyone")


PyApp()
```

```
gtk.main()
```

We display part of the lyrics from the Natasha Bedingfields Soulmate song.

```
  cr.select_font_face("Purisa", cairo.FONT_SLANT_NORMAL,
      cairo.FONT_WEIGHT_NORMAL)
```

Here we specify the font, that we use.

```
 cr.set_font_size(13)
```

We specify the size of the font.

```
 cr.move_to(20, 30)
```

We move to the point, where we will draw the text.

```
 cr.show_text("Most relationships seem so transitory")
```

The **show_text()** method draws text onto the window.

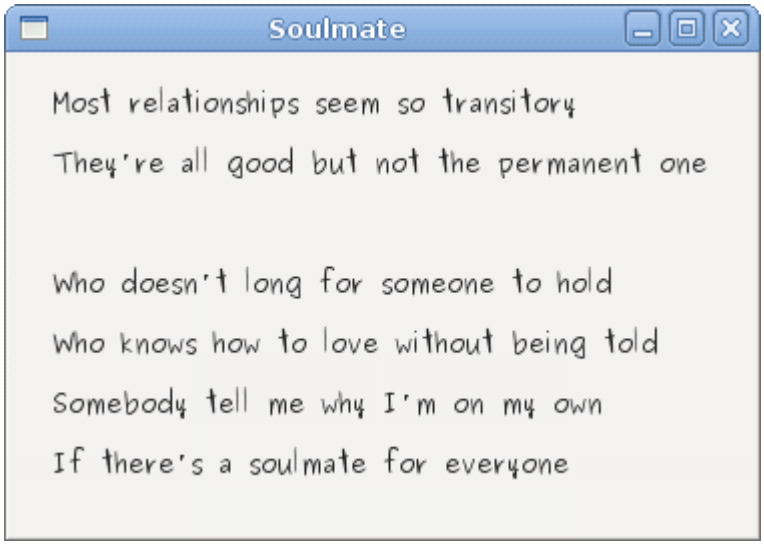


Figure: Soulmate

In this chapter of the PyGTK programming library, we were drawing with the Cairo graphics library.

Drawing with Cairo II

In this part of the PyGTK programming tutorial, we will continue drawing with the Cairo library.

Donut

In the following example we create an complex shape by rotating a bunch of ellipses.

donut.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program creates a donut
# with cairo library
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import math

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Donut")
        self.set_size_request(350, 250)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):
```

```
        cr = widget.window.cairo_create()

        cr.set_line_width(0.5)

        w = self.allocation.width
        h = self.allocation.height

        cr.translate(w/2, h/2)
        cr.arc(0, 0, 120, 0, 2*math.pi)
        cr.stroke()


        for i in range(36):
            cr.save()
            cr.rotate(i*math.pi/36)
            cr.scale(0.3, 1)
            cr.arc(0, 0, 120, 0, 2*math.pi)
            cr.restore()
            cr.stroke()


PyApp()
gtk.main()
```

In this example, we create a donut. The shape resembles a cookie, hence the name donut.

```
 cr.translate(w/2, h/2)
 cr.arc(0, 0, 120, 0, 2*math.pi)
 cr.stroke()
```

In the beginning there is an ellipse.

```
  for i in range(36):
      cr.save()
      cr.rotate(i*math.pi/36)
      cr.scale(0.3, 1)
      cr.arc(0, 0, 120, 0, 2*math.pi)
      cr.restore()
      cr.stroke()
```

After several rotations, there is a donut. We insulate each rotate and scale operations from one another with the **save()** and **restore()** methods.

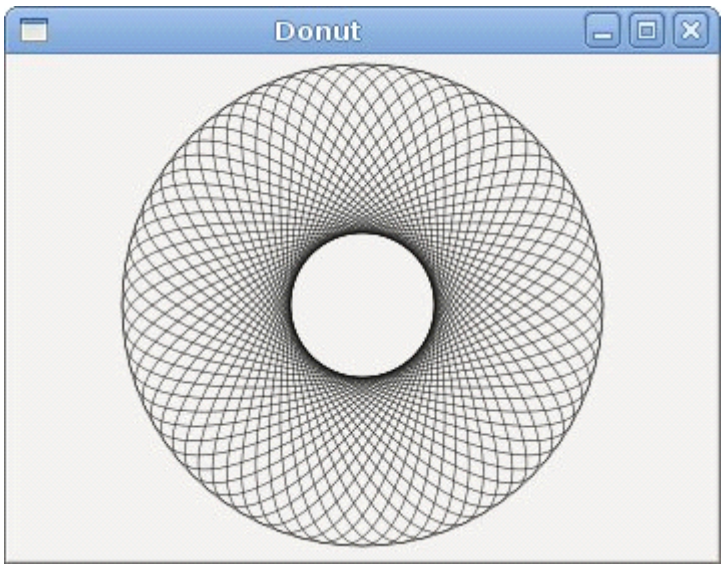


Figure: Donut

Gradients

In computer graphics, gradient is a smooth blending of shades from light to dark or from one color to another. In 2D drawing programs and paint programs, gradients are used to create colorful backgrounds and special effects as well as to simulate lights and shadows. (answers.com)

gradients.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program works with
# gradients in cairo
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import gtk
import cairo

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()
```

```
        self.set_title("Gradients")
        self.set_size_request(340, 390)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()
        lg1 = cairo.LinearGradient(0.0, 0.0, 350.0, 350.0)

        count = 1

        i = 0.1
        while i < 1.0:
            if count % 2:
                lg1.add_color_stop_rgba(i, 0, 0, 0, 1)
            else:
                lg1.add_color_stop_rgba(i, 1, 0, 0, 1)
            i = i + 0.1
            count = count + 1


        cr.rectangle(20, 20, 300, 100)
        cr.set_source(lg1)
        cr.fill()

        lg2 = cairo.LinearGradient(0.0, 0.0, 350.0, 0)

        count = 1

        i = 0.05
        while i < 0.95:
            if count % 2:
                lg2.add_color_stop_rgba(i, 0, 0, 0, 1)
            else:
                lg2.add_color_stop_rgba(i, 0, 0, 1, 1)
```

```
            i = i + 0.025
            count = count + 1

        cr.rectangle(20, 140, 300, 100)
        cr.set_source(lg2)
        cr.fill()

        lg3 = cairo.LinearGradient(20.0, 260.0,  20.0, 360.0)
        lg3.add_color_stop_rgba(0.1, 0, 0, 0, 1)
        lg3.add_color_stop_rgba(0.5, 1, 1, 0, 1)
        lg3.add_color_stop_rgba(0.9, 0, 0, 0, 1)

        cr.rectangle(20, 260, 300, 100)
        cr.set_source(lg3)
        cr.fill()


PyApp()
gtk.main()
```

In our example, we draw three rectangles with three different gradients.

```
lg1 = cairo.LinearGradient(0.0, 0.0, 350.0, 350.0)
```

Here we create a linear gradient pattern. The parameters specify the line, along which we draw the gradient. In our case it is a vertical line.

```
 lg3 = cairo.LinearGradient(20.0, 260.0,  20.0, 360.0)
 lg3.add_color_stop_rgba(0.1, 0, 0, 0, 1)
 lg3.add_color_stop_rgba(0.5, 1, 1, 0, 1)
 lg3.add_color_stop_rgba(0.9, 0, 0, 0, 1)
```

We define color stops to produce our gradient pattern. In this case, the gradient is a blending of black and yellow colors. By adding two black and one yellow stops, we create a horizontal gradient pattern. What do these stops actually mean? In our case, we begin with black color, which will stop at 1/10 of the size. Then we begin to gradually paint in yellow, which will culminate at the centre of the shape. The yellow color stops at 9/10 of the size, where we begin painting in black again, until the end.

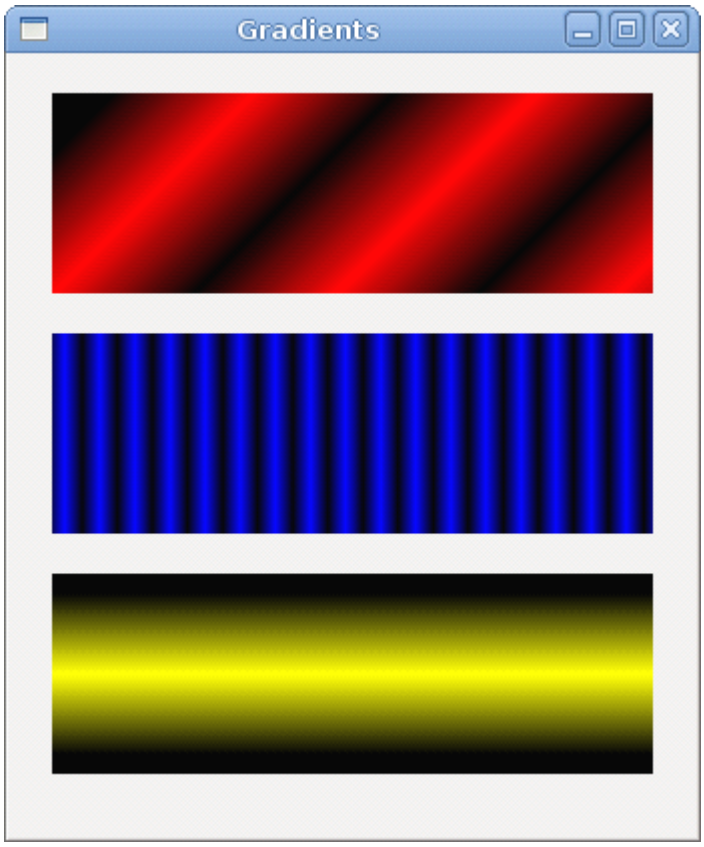


Figure: Gradients

Puff

In the following example, we create a puff effect. The example will display a growing centered text, that will gradually fade out from some point. This is a very common effect, which you can often see in flash animations.

puff.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program creates a puff
# effect
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009
```

```
import gtk
import glib
import cairo


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Puff")
        self.resize(350, 200)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        self.darea = gtk.DrawingArea()
        self.darea.connect("expose-event", self.expose)
        self.add(self.darea)

        self.timer = True
        self.alpha = 1.0
        self.size = 1.0

        glib.timeout_add(14, self.on_timer)

        self.show_all()

    def on_timer(self):
        if not self.timer: return False

        self.darea.queue_draw()
        return True


    def expose(self, widget, event):

        cr = widget.window.cairo_create()

        w = self.allocation.width
        h = self.allocation.height

        cr.set_source_rgb(0.5, 0, 0)
        cr.paint()
```

```
        cr.select_font_face("Courier", cairo.FONT_SLANT_NORMAL,
cairo.FONT_WEIGHT_BOLD)

        self.size = self.size + 0.8

        if self.size > 20:
            self.alpha = self.alpha - 0.01

        cr.set_font_size(self.size)
        cr.set_source_rgb(1, 1, 1)

        (x, y, width, height, dx, dy) = cr.text_extents("ZetCode")

        cr.move_to(w/2 - width/2, h/2)
        cr.text_path("ZetCode")
        cr.clip()
        cr.stroke()
        cr.paint_with_alpha(self.alpha)

        if self.alpha <= 0:
            self.timer = False


PyApp()
gtk.main()
```

The example creates a growing and fading text on the window.

```
 glib.timeout_add(14, self.on_timer)
```

Every 14 ms the **on_timer()** method is called.

```
 def on_timer(self):
     if not self.timer: return False

     self.darea.queue_draw()
     return True
```

In the **on_timer()** method, we call the **queue_draw()** method upon the drawing area, which triggers the expose signal.

```
 cr.set_source_rgb(0.5, 0, 0)
 cr.paint()
```

We set the background color to dark red color.

```
self.size = self.size + 0.8
```

Each cycle, the font size will grow by 0.8 units.

```
if self.size > 20:
    self.alpha = self.alpha - 0.01
```

The fading out begins after the font size is bigger than 20.

```
(x, y, width, height, dx, dy) = cr.text_extents("ZetCode")
```

We get the text metrics.

```
cr.move_to(w/2 - width/2, h/2)
```

We use the text metrics to center the text on the window.

```
cr.text_path("ZetCode")
cr.clip()
```

We get the path of the text and set the current clip region to it.

```
cr.stroke()
cr.paint_with_alpha(self.alpha)
```

We paint the current path and take alpha value into account.

Figure: Puff

Reflection

In the next example we show a reflected image. This beautiful effect makes an illusion as if the image was reflected in water.

reflection.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# ZetCode PyGTK tutorial
#
# This program creates an
# image reflection
#
# author: Jan Bodnar
# website: zetcode.com
# last edited: April 2011


import gtk
import cairo
import sys

class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Reflection")
        self.resize(300, 350)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        darea = gtk.DrawingArea()
        darea.connect("expose-event", self.expose)
        self.add(darea)

        try:
            self.surface =
cairo.ImageSurface.create_from_png("slanec.png")
        except Exception, e:
```

```
            print e.message
            sys.exit(1)


        self.imageWidth = self.surface.get_width()
        self.imageHeight = self.surface.get_height()
        self.gap = 40
        self.border = 20

        self.show_all()

    def expose(self, widget, event):

        cr = widget.window.cairo_create()


        w = self.allocation.width
        h = self.allocation.height

        lg = cairo.LinearGradient(w/2, 0, w/2, h*3)
        lg.add_color_stop_rgba(0, 0, 0, 0, 1)
        lg.add_color_stop_rgba(h, 0.2, 0.2, 0.2, 1)

        cr.set_source(lg)
        cr.paint()

        cr.set_source_surface(self.surface, self.border, self.border)
        cr.paint()

        alpha = 0.7
        step = 1.0 / self.imageHeight

        cr.translate(0, 2 * self.imageHeight + self.gap)
        cr.scale(1, -1)

        i = 0


        while(i < self.imageHeight):

            cr.rectangle(self.border, self.imageHeight-i,
self.imageWidth, 1)

            i = i + 1
```

```
            cr.save()
            cr.clip()
            cr.set_source_surface(self.surface, self.border,
self.border)
            alpha = alpha - step
            cr.paint_with_alpha(alpha)
            cr.restore()


PyApp()
gtk.main()
```

The example shows a reflected castle.

```
 lg = cairo.LinearGradient(w/2, 0, w/2, h*3)
 lg.add_color_stop_rgba(0, 0, 0, 0, 1)
 lg.add_color_stop_rgba(h, 0.2, 0.2, 0.2, 1)

 cr.set_source(lg)
 cr.paint()
```

The background is filled with a gradiet paint. The paint is a smooth blending from black to dark gray.

```
 cr.translate(0, 2 * self.imageHeight + self.gap)
 cr.scale(1, -1)
```

This code flips the image and translates it below the original image. The translation operation is necessary, because the scaling operation makes the image upside down and translates the image up. To understand what happens, simply take a photograph and place it on the table. Now flip it.

```
cr.rectangle(self.border, self.imageHeight-i, self.imageWidth, 1)

i = i + 1

cr.save()
cr.clip()
cr.set_source_surface(self.surface, self.border, self.border)
alpha = alpha - step
cr.paint_with_alpha(alpha)
cr.restore()
```

Crucial part of the code. We make the second image transparent. But the transparency is not constant. The image gradually fades out. This is achieved with the gradient.

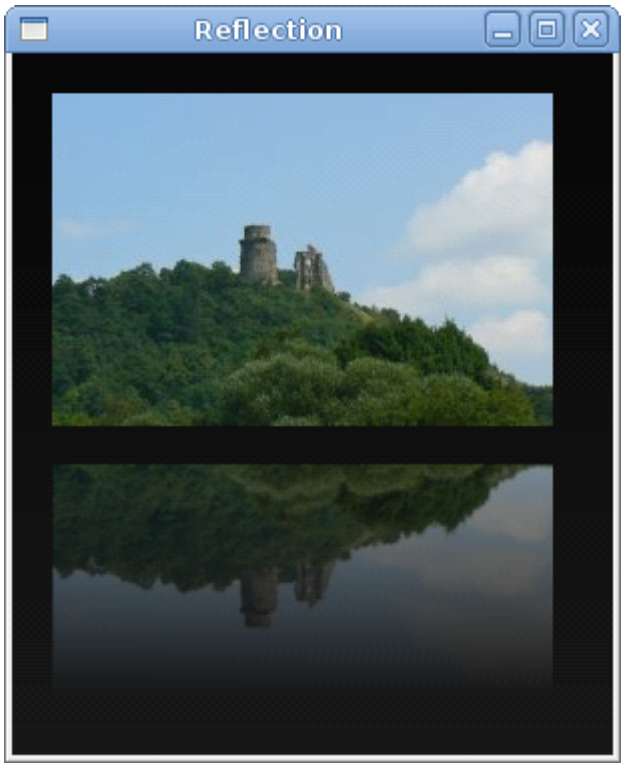


Figure: Reflection

Waiting

In this examle, we use transparency effect to create a waiting demo. We will draw 8 lines that will gradually fade out creating an illusion, that a line is moving. Such effects are often used to inform users, that a lengthy task is going on behind the scenes. An example is streaming video over the internet.

waiting.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This program creates an
# waiting effect
#
```

```
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009


import gtk
import glib
import math
import cairo


trs = (
    ( 0.0, 0.15, 0.30, 0.5, 0.65, 0.80, 0.9, 1.0 ),
    ( 1.0, 0.0,  0.15, 0.30, 0.5, 0.65, 0.8, 0.9 ),
    ( 0.9, 1.0,  0.0,  0.15, 0.3, 0.5, 0.65, 0.8 ),
    ( 0.8, 0.9,  1.0,  0.0,  0.15, 0.3, 0.5, 0.65 ),
    ( 0.65, 0.8, 0.9,  1.0,  0.0,  0.15, 0.3, 0.5 ),
    ( 0.5, 0.65, 0.8, 0.9, 1.0,  0.0,  0.15, 0.3 ),
    ( 0.3, 0.5, 0.65, 0.8, 0.9, 1.0,  0.0,  0.15 ),
    ( 0.15, 0.3, 0.5, 0.65, 0.8, 0.9, 1.0,  0.0, )
)


class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Waiting")
        self.set_size_request(250, 150)
        self.set_position(gtk.WIN_POS_CENTER)

        self.connect("destroy", gtk.main_quit)

        self.darea = gtk.DrawingArea()
        self.darea.connect("expose-event", self.expose)
        self.add(self.darea)

        self.count = 0

        glib.timeout_add(100, self.on_timer)

        self.show_all()
```

```
    def on_timer(self):
        self.count = self.count + 1
        self.darea.queue_draw()
        return True


    def expose(self, widget, event):

        cr = widget.window.cairo_create()

        cr.set_line_width(3)
        cr.set_line_cap(cairo.LINE_CAP_ROUND)

        w = self.allocation.width
        h = self.allocation.height

        cr.translate(w/2, h/2)

        for i in range(8):
            cr.set_source_rgba(0, 0, 0, trs[self.count%8][i])
            cr.move_to(0.0, -10.0)
            cr.line_to(0.0, -40.0)
            cr.rotate(math.pi/4)
            cr.stroke()


PyApp()
gtk.main()
```

We draw eight lines with eight different alpha values.

```
 glib.timeout_add(100, self.on_timer)
```

We use a timer function to create animation.

```
trs = (
    ( 0.0, 0.15, 0.30, 0.5, 0.65, 0.80, 0.9, 1.0 ),
    ...
)
```

This is a two dimensional tuple of transparency values used in this demo. There are 8 rows, each for one state. Each of the 8 lines will continuosly use these values.

```
cr.set_line_width(3)
cr.set_line_cap(cairo.LINE_CAP_ROUND)
```

We make the lines a bit thicker, so that they are better visible. We draw the lines with rouded caps. They look then better.

```
cr.set_source_rgba(0, 0, 0, trs[self.count%8][i]
```

Here we define the transparency value for a line.

```
cr.move_to(0.0, -10.0)
cr.line_to(0.0, -40.0)
cr.rotate(math.pi/4)
cr.stroke()
```

These code lines will draw each of the eight lines.

Figure: Waiting

In this chapter of the PyGTK programming library, we did some more advanced drawing with the Cairo library.

Snake game in PyGTK

In this part of the PyGTK programming tutorial, we will create a Snake game clone.

Snake game

Snake is an older classic video game. It was first created in late 70s. Later it was brought to PCs. In this game the player controls a snake. The objective is to eat as many apples as possible. Each time the snake eats an apple, its body grows. The snake must avoid the walls and its own body. This game is sometimes called Nibbles.

Development

The size of each of the joints of a snake is 10px. The snake is controlled with the cursor keys. Initially the snake has three joints. The game starts immediately. If the game is finished, we display "Game Over" message in the middle of the Board.

snake.py

```
#!/usr/bin/python

# ZetCode PyGTK tutorial
#
# This is a simple snake game
# clone
#
# author: jan bodnar
# website: zetcode.com
# last edited: February 2009

import sys
import gtk
import cairo
import random
import glib


WIDTH = 300
HEIGHT = 270
DOT_SIZE = 10
ALL_DOTS = WIDTH * HEIGHT / (DOT_SIZE * DOT_SIZE)
RAND_POS = 26

x = [0] * ALL_DOTS
y = [0] * ALL_DOTS
```

```
class Board(gtk.DrawingArea):

    def __init__(self):
        super(Board, self).__init__()

        self.modify_bg(gtk.STATE_NORMAL, gtk.gdk.Color(0, 0, 0))
        self.set_size_request(WIDTH, HEIGHT)

        self.connect("expose-event", self.expose)

        self.init_game()

    def on_timer(self):

        if self.inGame:
            self.check_apple()
            self.check_collision()
            self.move()
            self.queue_draw()
            return True
        else:
            return False

    def init_game(self):

        self.left = False
        self.right = True
        self.up = False
        self.down = False
        self.inGame = True
        self.dots = 3

        for i in range(self.dots):
            x[i] = 50 - i * 10
            y[i] = 50

        try:
            self.dot = cairo.ImageSurface.create_from_png("dot.png")
            self.head = cairo.ImageSurface.create_from_png("head.png")
            self.apple =
cairo.ImageSurface.create_from_png("apple.png")
        except Exception, e:
```

```
            print e.message
            sys.exit(1)

        self.locate_apple()
        glib.timeout_add(100, self.on_timer)




    def expose(self, widget, event):

        cr = widget.window.cairo_create()

        if self.inGame:
            cr.set_source_rgb(0, 0, 0)
            cr.paint()

            cr.set_source_surface(self.apple, self.apple_x,
self.apple_y)
            cr.paint()

            for z in range(self.dots):
                if (z == 0):
                    cr.set_source_surface(self.head, x[z], y[z])
                    cr.paint()
                else:
                    cr.set_source_surface(self.dot, x[z], y[z])
                    cr.paint()
        else:
            self.game_over(cr)



    def game_over(self, cr):

        w = self.allocation.width / 2
        h = self.allocation.height / 2

        (x, y, width, height, dx, dy) = cr.text_extents("Game Over")

        cr.set_source_rgb(65535, 65535, 65535)
        cr.move_to(w - width/2, h)
        cr.show_text("Game Over")
        self.inGame = False
```

```
def check_apple(self):

    if x[0] == self.apple_x and y[0] == self.apple_y:
        self.dots = self.dots + 1
        self.locate_apple()


def move(self):

    z = self.dots

    while z > 0:
        x[z] = x[(z - 1)]
        y[z] = y[(z - 1)]
        z = z - 1

    if self.left:
        x[0] -= DOT_SIZE

    if self.right:
        x[0] += DOT_SIZE

    if self.up:
        y[0] -= DOT_SIZE

    if self.down:
        y[0] += DOT_SIZE



def check_collision(self):

    z = self.dots

    while z > 0:
        if z > 4 and x[0] == x[z] and y[0] == y[z]:
            self.inGame = False
        z = z - 1

    if y[0] > HEIGHT - DOT_SIZE:
        self.inGame = False
```

```
        if y[0] < 0:
            self.inGame = False

        if x[0] > WIDTH - DOT_SIZE:
            self.inGame = False

        if x[0] < 0:
            self.inGame = False


    def locate_apple(self):

        r = random.randint(0, RAND_POS)
        self.apple_x = r * DOT_SIZE
        r = random.randint(0, RAND_POS)
        self.apple_y = r * DOT_SIZE


    def on_key_down(self, event):

        key = event.keyval

        if key == gtk.keysyms.Left and not self.right:
            self.left = True
            self.up = False
            self.down = False


        if key == gtk.keysyms.Right and not self.left:
            self.right = True
            self.up = False
            self.down = False


        if key == gtk.keysyms.Up and not self.down:
            self.up = True
            self.right = False
            self.left = False


        if key == gtk.keysyms.Down and not self.up:
            self.down = True
            self.right = False
```

```
            self.left = False


class Snake(gtk.Window):

    def __init__(self):
        super(Snake, self).__init__()

        self.set_title('Snake')
        self.set_size_request(WIDTH, HEIGHT)
        self.set_resizable(False)
        self.set_position(gtk.WIN_POS_CENTER)

        self.board = Board()
        self.connect("key-press-event", self.on_key_down)
        self.add(self.board)

        self.connect("destroy", gtk.main_quit)
        self.show_all()


    def on_key_down(self, widget, event):

        key = event.keyval
        self.board.on_key_down(event)


Snake()
gtk.main()
```

First we will define some globals used in our game.

The **WIDTH** and **HEIGHT** constants determine the size of the Board. The **DOT_SIZE** is the size of the apple and the dot of the snake. The **ALL_DOTS** constant defines the maximum number of possible dots on the Board. The **RAND_POS** constant is used to calculate a random position of an apple. The **DELAY** constant determines the speed of the game.

```
 x = [0] * ALL_DOTS
 y = [0] * ALL_DOTS
```

These two lists store x, y coordinates of all possible joints of a snake.

The **init_game()** method initializes variables, loads images and starts a timeout function.

```
self.left = False
self.right = True
self.up = False
self.down = False
self.inGame = True
self.dots = 3
```

When the game starts, the snake has three joints. And it is heading to the right.

In the **move()** method we have the key algorithm of the game. To understand it, look at how the snakeis moving. You control the head of the snake. You can change its direction with the cursor keys. The rest of the joints move one position up the chain. The second joint moves where the first was, the third joint where the second was etc.

```
while z > 0:
    x[z] = x[(z - 1)]
    y[z] = y[(z - 1)]
    z = z - 1
```

This code moves the joints up the chain.

```
if self.left:
    x[0] -= DOT_SIZE
```

Move the head to the left.

In the **checkCollision()** method, we determine if the snake has hit itself or one of the walls.

```
while z > 0:
    if z > 4 and x[0] == x[z] and y[0] == y[z]:
        self.inGame = False
    z = z - 1
```

Finish the game, if the snake hits one of its joints with the head.

```
if y[0] > HEIGHT - DOT_SIZE:
    self.inGame = False
```

Finish the game, if the snake hits the bottom of the Board.

The **locate_apple()** method locates an apple randomly on the form.

```
r = random.randint(0, RAND_POS)
```

We get a random number from 0 to RAND_POS - 1.

```
self.apple_x = r * DOT_SIZE
...
self.apple_y = r * DOT_SIZE
```

These line set the x, y coordinates of the apple object.

```
    self.connect("key-press-event", self.on_key_down)
    ...

def on_key_down(self, widget, event):

    key = event.keyval
    self.board.on_key_down(event)
```

We catch the key press event in the Snake class, and delegate the processing to the board object.

In the **on_key_dow()** method of the Board class, we determime which keys the player hit.

```
if key == gtk.keysyms.Left and not self.right:
    self.left = True
    self.up = False
    self.down = False
```

If we hit the left cursor key, we set **self.left** variable to True. This variable is used in the **move()** method to change coordinates of the snake object. Notice also, that when the snake is heading to the right, we cannot turn immediately to the left.

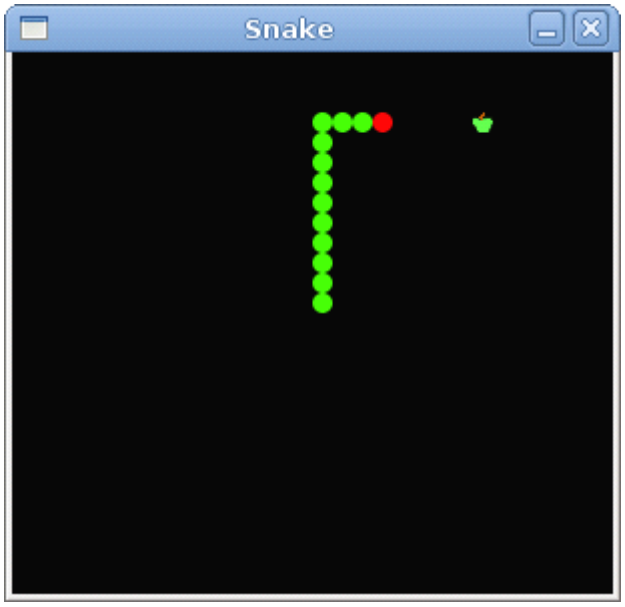


Figure: Snake

This was the Snake computer game programmed using PyGTK programming library.

Custom widget in PyGTK

Have you ever looked at an application and wondered, how a particular gui item was created? Probably every wannabe programmer has. Then you were looking at a list of widgets provided by your favourite gui library. But you couldn't find it. Toolkits usually provide only the most common widgets like buttons, text widgets, sliders etc. No toolkit can provide all possible widgets.

There are actually two kinds of toolkits. Spartan toolkits and heavy weight toolkits. The FLTK toolkit is a kind of a spartan toolkit. It provides only the very basic widgets and assumes, that the programemer will create the more complicated ones himself. wxWidgets is a heavy weight one. It has lots of widgets. Yet it does not provide the more specialized widgets. For example a speed meter widget, a widget that measures the capacity of a CD to be burned (found e.g. in nero). Toolkits also don't have usually charts.

Programmers must create such widgets by themselves. They do it by using the drawing tools provided by the toolkit. There are two possibilities.

A programmer can modify or enhance an existing widget. Or he can create a custom widget from scratch.

Burning widget

This is an example of a widget, that we create from scratch. This widget can be found in various media burning applications, like Nero Burning ROM.

burning.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

# ZetCode PyGTK tutorial
#
# This example creates a burning
# custom widget
#
# author: Jan Bodnar
# website: zetcode.com
# last edited: April 2011


import gtk
import cairo

class Burning(gtk.DrawingArea):

    def __init__(self, parent):

        self.par = parent
        super(Burning, self).__init__()

        self.num = ( "75", "150", "225", "300",
            "375", "450", "525", "600", "675" )

        self.set_size_request(-1, 30)
        self.connect("expose-event", self.expose)


    def expose(self, widget, event):

        cr = widget.window.cairo_create()
        cr.set_line_width(0.8)
```

```
cr.select_font_face("Courier",
    cairo.FONT_SLANT_NORMAL, cairo.FONT_WEIGHT_NORMAL)
cr.set_font_size(11)

width = self.allocation.width

self.cur_width = self.par.get_cur_value()

step = round(width / 10.0)

till = (width / 750.0) * self.cur_width
full = (width / 750.0) * 700

if (self.cur_width >= 700):

    cr.set_source_rgb(1.0, 1.0, 0.72)
    cr.rectangle(0, 0, full, 30)
    cr.save()
    cr.clip()
    cr.paint()
    cr.restore()

    cr.set_source_rgb(1.0, 0.68, 0.68)
    cr.rectangle(full, 0, till-full, 30)
    cr.save()
    cr.clip()
    cr.paint()
    cr.restore()

else:
    cr.set_source_rgb(1.0, 1.0, 0.72)
    cr.rectangle(0, 0, till, 30)
    cr.save()
    cr.clip()
    cr.paint()
    cr.restore()


cr.set_source_rgb(0.35, 0.31, 0.24)

for i in range(1, len(self.num) + 1):
    cr.move_to(i*step, 0)
    cr.line_to(i*step, 5)
```

```
            cr.stroke()

            (x, y, width, height, dx, dy) =
cr.text_extents(self.num[i-1])
            cr.move_to(i*step-width/2, 15)
            cr.text_path(self.num[i-1])
            cr.stroke()



class PyApp(gtk.Window):

    def __init__(self):
        super(PyApp, self).__init__()

        self.set_title("Burning")
        self.set_size_request(350, 200)
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)

        self.cur_value = 0

        vbox = gtk.VBox(False, 2)

        scale = gtk.HScale()
        scale.set_range(0, 750)
        scale.set_digits(0)
        scale.set_size_request(160, 40)
        scale.set_value(self.cur_value)
        scale.connect("value-changed", self.on_changed)

        fix = gtk.Fixed()
        fix.put(scale, 50, 50)

        vbox.pack_start(fix)

        self.burning = Burning(self)
        vbox.pack_start(self.burning, False, False, 0)

        self.add(vbox)
        self.show_all()


    def on_changed(self, widget):
```

```
        self.cur_value = widget.get_value()
        self.burning.queue_draw()


    def get_cur_value(self):
        return self.cur_value


PyApp()
gtk.main()
```

We put a **DrawingArea** on the bottom of the window and draw the entire widget manually. All the important code resides in the **expose()** method of the Burning class. This widget shows graphically the total capacity of a medium and the free space available to us. The widget is controlled by a scale widget. The minimum value of our custom widget is 0, the maximum is 750. If we reach value 700, we began drawing in red colour. This normally indicates overburning.

```
self.num = ( "75", "150", "225", "300",
    "375", "450", "525", "600", "675" )
```

These numbers are shown on the burning widget. They show the capacity of the medium.

```
self.cur_width = self.par.get_cur_value()
```

These two lines get the current number from the scale widget. We get the parent widget and from the parent widget, we get the current value.

```
till = (width / 750.0) * self.cur_width
full = (width / 750.0) * 700
```

The till parameter determines the total size to be drawn. This value comes from the slider widget. It is a proportion of the whole area. The full parameter determines the point, where we begin to draw in red color.

```
cr.set_source_rgb(1.0, 1.0, 0.72)
cr.rectangle(0, 0, till, 30)
cr.save()
cr.clip()
cr.paint()
cr.restore()
```

This code here, draws a yellow rectangle up to point, where the medium is full.

```
(x, y, width, height, dx, dy) = cr.text_extents(self.num[i-1])
cr.move_to(i*step-width/2, 15)
cr.text_path(self.num[i-1])
cr.stroke()
```

This code here draws the numbers on the burning widget. We calculate the **TextExtents** to position the text correctly.

```
def on_changed(self, widget):
    self.cur_value = widget.get_value()
    self.burning.queue_draw()
```

We get the value from the scale widget, store it in the **cur_value** variable for later use. We redraw the burning widget.

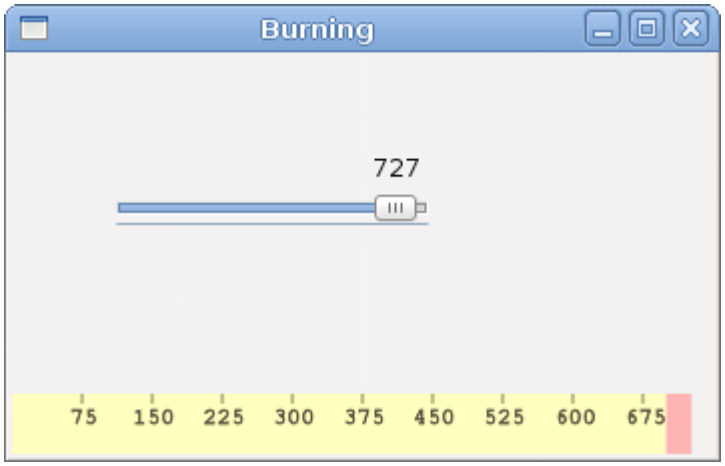


Figure: Burning widget

In this chapter, we created a custom widget in PyGTK.