The wxPython tutorial

The wxPython tutorial is the largest and most advanced wxPython tutorial available on the Internet. Suitable for beginners and intermediate programmers.

Introduction to wxPython

An Application

An **application** is a computer program that performs a specific task or a group of tasks. Web browser, media player, word processor are examples of typical applications. A term **tool** or **utility** is used for a rather small and simple application that performs a single task. A unix cp program is an example of a such a tool. All these together form **computer software**. Computer software is the broadest term used to describe the operating system, data, computer programs, applications, mp3 files or computer games. Applications can be created for four different areas.

Online shopping applications, wikis, weblogs are examples of popular web applications. They are accessed with a web browser. Examples of desktop applications include Maya, Opera, Open Office or Winamp. Enterprise computing is a specific area. Applications in these area are complex and large. Applications created for portables include all programs developed for mobile phones, communicators, pda's and similar.

Programming languages

There are currently several widely used programming languages. The following list is based on the TIOBE Programming Community Index. The numbers are from May 2007.

Position	Language	Ratings

1	Java	19.1%
2	C	15.2%
3	C++	10.1%
4	PHP	8.7%
5	Visual Basic	8.4%
6	Perl	6.2%
7	Python	3.8%
8	C#	3.7%
9	JavaScript	3.1%
10	Ruby	2.6%

Java is the most widely used programming language. Java excels in creating portable mobile applications, programming various appliances and in creating enterprise applications. Every fourth application is programmed in C/C++. They are standard for creating operating systems and various desktop applications. C/C++ are the most widely used system programming languages. Most famous desktop applications were created in C++. May it be MS Office, Macromedia Flash, Adobe Photoshop or 3D Max. These two languages also dominate the game programming business.

PHP dominates over the Web. While Java is used mainly by large organizations, PHP is used by smaller companies and individuals. PHP is used to create dynamic web applications. Visual Basic is mainly used in **RAD**. RAD stands for rapid application development.

Perl, Python and Ruby are the most widely used scripting languages. They share many similarities. They are close competitors.

The time of the C# has still not come yet. It was planned to be the next big language. Javascript is a client side programming language, which runs in a browser. It is a de facto standard language and has no competition in it's area.

Python

Python is a successful scripting language. It was initially developed by **Guido van Rossum**. It was first

released in 1991. Python was inspired by ABC and Haskell programming languages. Python is a high level, general purpose, multiplatform, interpreted language. Some prefer to call it a dynamic language. It is easy to learn. Python is a minimalistic language. One of it's most visible features is that it does not use semicolons nor brackets. Python uses intendation instead. The most recent version of python is 2.5, which was released in September 2006. Today, Python is maintained by a large group of volunteers worldwide.

For creating graphical user interfaces, python programmers can choose among three decent options. PyGTK, wxPython and PyQt. The offcial python "toolkit" is TkInter. It is slow, looks terrible on all platforms, it has not been updated for ages. It also depends on Tcl (the tcl language must be included), which is odd. All of the above mentioned toolkits are superior to it. It remains a mystery, why it was not excluded years ago.

wxPython

wxPython is a cross platform toolkit for creating desktop GUI applications. With wxPython developers can create applications on Windows, Mac and on various Unix systems. wxPython is a wrapper around wxWidgets, which is a mature cross platform C++ library. wxPython consists of the five basic modules.

Controls module provides the common widgets found in graphical applications. For example a Button, a Toolbar, or a Notebook. Widgets are called controls under Windows OS. The **Core** module consists of elementary classes, that are used in development. These classes include the Object class, which is the mother of all classes, Sizers, which are used for widget layout, Events, basic geometry classses like Point and Rectangle. The Graphics Device Interface (**GDI**) is a set of classes used for drawing onto the widgets. This module contains classes for manipulation of Fonts, Colours, Brushes, Pens or Images. The **Misc** module contains of various other classes and module functions. These classes are used for logging, application configuration, system settings, working with display or joystick. The **Windows** module consists of various windows, that form an application. Panel, Dialog, Frame or Scrolled Window.

wxPython API

wxPython API is a set of functions and widgets. Widgets are essential building blocks of a GUI application. Under Windows widgets are calles controls. We can roughly divide programmers into two groups. They code applications or libraries. In our case, wxPython is a library that is used by application programmers to code applications. Technically, wxPython is a wrapper over a C++ GUI API called wxWidgets. So it is not a native API. e.g. not written directly in Python. The only native GUI library for an interpreted language that I know is Java's Swing library.

In wxPython we have lot's of widgets. These can be divided into some logical groups.

Base Widgets

These widgets provide basic functionality for derived widgets. They are called ancestors. They are usually not used directly.

Top level Widgets

These widgets exist independently of each other.

Containers

Containers contain other widgets.

Dynamic Widgets

These widgets can be edited by users.

Static Widgets

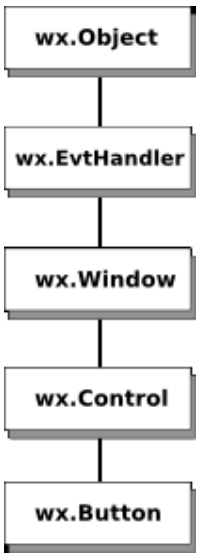
These widgets display informatin. They cannot be edited by user.

Other Widgets

These widgets implement statusbar, toolbar and menubar in an application.

Inheritance

There is a specific relation among widgets in wxPython. This relation is developed by inheritance. The inheritance is a crucial part of the object oriented programming. Widgets form a hierarchy. Widgets can inherit functionality from other widgets. Existing classes are called base classes, parents, or ancestors. The widgets that inherit we call derived widgets, child widgets or descendants. The terminology is borrowed from biology.



Inheritance of a button

Say we use a button widget in our application. The button widget inherits from 4 different base classes. The closest class is the *wx.Control* class. A button widget is a kind of a small window. All widgets that appear on the screen are windows. Therefore they inherit from *wx.Window* class. There are objects that are invisible. Examples are sizers, device context or locale object. There are also classes that are visible but they are not windows. For example a color object, caret object or a cursor object. Not all widgets are controls. For example *wx.Dialog* is not a kind of control. The controls are widgets

that are placed on other widgets called **containers**. That's why we have a separate *wx.Control* base class.

Every window can react to events. So does the button widget. By clicking on the button, we launch the *wx.EVT_COMMAND_BUTTON_CLICKED* event. The button widget inherits the *wx.EvtHandler* via the *wx.Window* class. Each widget that reacts to events must inherit from *wx.EvtHandler* class. Finally all objects inherit from *wx.Object* class. This is the Eve, mother of all objets in wxPython.

Setting up wxPython

In this section we will show how to set up wxPython library. The process is very easy provided you have the right OS.

Setting up wxPython on Windows XP.

We can download binaries from the wxPython official web site. We must choose the right binaries depending on our python version. There are two basic options.

- win32-unicode
- win32-ansi

Usually the unicode version is the choice. Unicode version supports other languages than english. The installer automatically finds the path to the python interpreter. The only thing we must do, is to check that we agree with the licence. That's all.

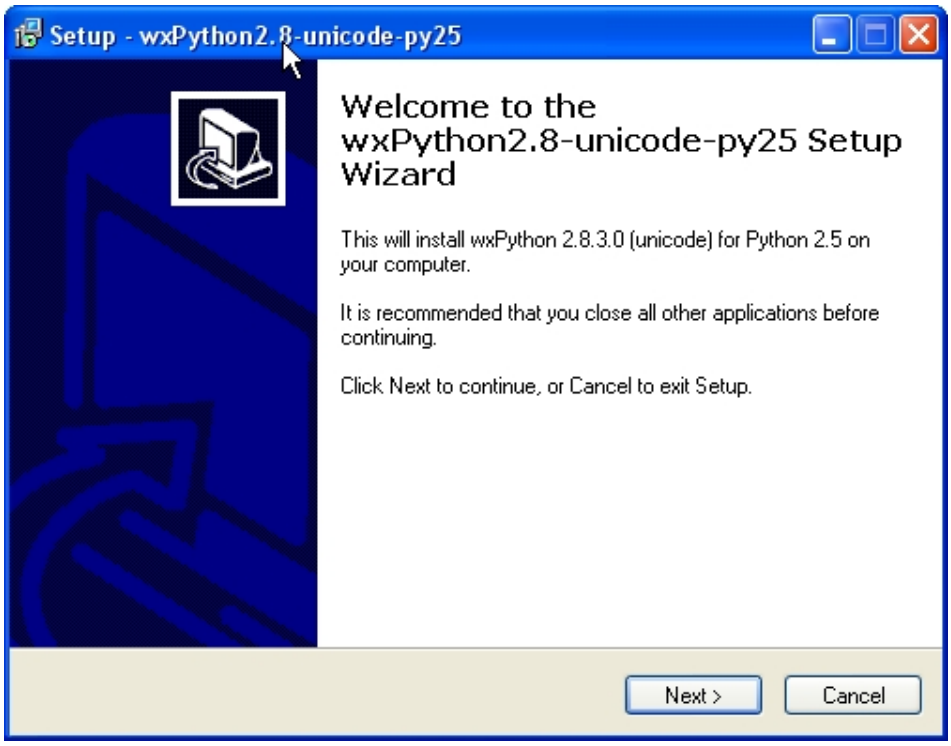


Figure: Installing wxPython on Windows XP

There is also another separate package. It is called *win32-docs-demos* It contains the famous demo example, documentation and examples. This is geared towards developers. One useful note regarding python interpreter on windows. If we click on the python script file a command line window pops up. This is the default behaviour. We can change this by associating python programs with the *pythonw.exe* file. Right click on the python icon. Choose properties. Press Change button. And choose the *pythonw.exe*, which is located in the python install directory. On my box it is C:\Program Files\Python25\pythonw.exe.

Setting up wxPython on Ubuntu Linux

The installation process is even simplier than on Windows XP. Ubuntu is a great linux distro. Installing wxPython is a piece of cake. The installation is done with the Synaptic Package Manager. We can find it under System - Administration menu. The wxPython package is called *python-wxgtkx.x*. The package manager automatically cares about all the necessary dependencies. We mark the package for installation and click apply. The wxPython library is downloaded and installed on our system. Packages on Ubuntu are pretty fresh. Other linux distros might have older packages. wxPython is a rapidly

evolving toolkit. To stay up to date, we can be forced to install wxPython from sources.

Home ‡ Contents ‡ Top of Page

First Steps

Simple example

We start with a very simple example. Our first script will only show a small window. It won't do much. We will analyze the script line by line. Here is the code:

```
#!/usr/bin/python

# simple.py

import wx

app = wx.App()

frame = wx.Frame(None, -1, 'simple.py')
frame.Show()

app.MainLoop()
```

```
#!/usr/bin/python

# simple.py
```

The first line is a she-bang followed by the path to a python interpreter. The second line is a comment. It provides a name for the script.

```
import wx
```

This line imports the basic wxPython modules. Namely the core, controls, gdi, misc and windows. Technically wx is a namespace. All functions and objects from the basic modules will start with a wx. prefix. The next line of code will create an application object.

```
app = wx.App()
```

Each wxPython program must have one application object.

```
frame = wx.Frame(None, -1, 'simple.py')
frame.Show()
```

Here we create a wx.Frame object. A wx.Frame widget is an important container widget. We will analyze this widget in detail later. The wx.Frame widget is a parent widget for other widgets. It has no parent itself. If we specify None for a parent parameter we indicate that our widget has no parents. It is a top widget in the hierarchy of widgets. After we create the wx.Frame widget, we must call the Show() method to actually display it on the screen.

```
app.MainLoop()
```

The last line enters the mainloop. The mainloop is an endless cycle. It catches and dispatches all events that exist during the life of our application.

This was a very simplistic example. Despite this simplicity we can do quite a lot with this window. We can resize the window, mazimize it, minimize it. This functionality requires a lot of coding. All this is hidden and provided by default by the wxPython toolkit. There is no reason for reinveinting the wheel.

Figure: simple.py

wx.Frame

wx.Frame widget is one of the most important widgets in wxPython. It is a container widget. It means that it can contain other widgets. Actually it can contain any window that is not a frame or dialog. wx.Frame consists of a title bar, borders and a central container area. The title bar and borders are optional. They can be removed by various flags.

wx.Frame has the following constructor. As we can see, it has seven parameters. The first parameter does not have a default value. The other six parameters do have. Those four parameters are optional. The first three are mandatory.

```
wx.Frame(wx.Window parent, int id=-1, string title='', wx.Point pos =
wx.Size size = wx.DefaultSize, style = wx.DEFAULT_FRAME_STYLE, string
```

wx.DEFAULT_FRAME_STYLE is a set of default flags. wx.MINIMIZE_BOX | wx.MAXIMIZE_BOX | wx.RESIZE_BORDER | wx.SYSTEM_MENU | wx.CAPTION | wx.CLOSE_BOX | wx.CLIP_CHILDREN. By combining various styles we can change the style of the wx.Frame widget. A short example follows.

```
#!/usr/bin/python

# nominimizebox.py

import wx

app = wx.App()
window = wx.Frame(None, style=wx.MAXIMIZE_BOX | wx.RESIZE_BORDER
        | wx.SYSTEM_MENU | wx.CAPTION |  wx.CLOSE_BOX)
window.Show(True)

app.MainLoop()
```

Our intention was to display a window without a mimimize box. So we did not specify this flag in the style parameter.

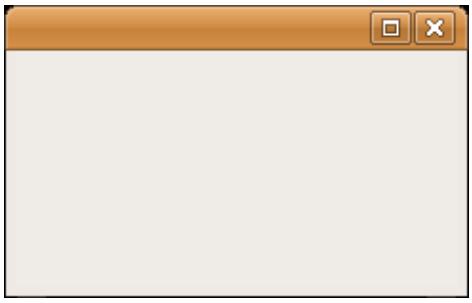


Figure: A window without a minimize box

Size and Position

We can specify the size of our application in two ways. We have a size parameter in the constructor of our widget. Or we can call the SetSize() method.

```
#!/usr/bin/python

# size.py

import wx

class Size(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 200))

        self.Show(True)


app = wx.App()
Size(None, -1, 'Size')
app.MainLoop()
```

```
wx.Frame.__init__(self, parent, id, title, size=(250, 200))
```

In the constructor we set the width of the wx.Frame widget to 250px. The height of the widget to 200px.

Similarly, we can position our application on the screen. By default the window is placed in the upper left corner of the screen. But it can differ on various OS platforms or even window managers. Some window managers place application windows themselves. Some of them do some optimalization, so that windows do not overlap. A programmer can position the window programatically. We

already saw a *pos* parameter in the constructor of our wx.Frame widget. By providing other than the default values, we can control the position ourselves.

Method	Description
Move(wx.Point point)	move a window to the given position
MoveXY(int x, int y)	move a window to the given position
SetPosition(wx.Point point)	set the position of a window
SetDimensions(wx.Point point, wx.Size size)	set the position and the size of a window

There are several methods to do this. Toss a coin.

```
#!/usr/bin/python

# move.py

import wx

class Move(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        self.Move((800, 250))
        self.Show(True)


app = wx.App()
Move(None, -1, 'Move')
app.MainLoop()
```

There is one particular situation. We might want to display our window maximized. In this case, the window is positioned at (0, 0) and takes the whole screen. wxPython internally calculates the screen coordinates. To maximize our wx.Frame, we call the *Maximize()* method. If we want to center our application on the screen, wxPython has a handy method. The *Centre()* method simply centers the window on the screen. No need to calculate the width and the height of the screen. Simply call the method.

```
#!/usr/bin/python

# centre.py

import wx

class Centre(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        self.Centre()
        self.Show(True)

app = wx.App()
Centre(None, -1, 'Centre')
app.MainLoop()
```

Widgets communicate

It is important to know, how widgets can communicate in application. Follow the next example.

```
#!/usr/bin/python

# communicate.py

import wx


class LeftPanel(wx.Panel):
    def __init__(self, parent, id):
        wx.Panel.__init__(self, parent, id, style=wx.BORDER_SUNKEN)

        self.text = parent.GetParent().rightPanel.text

        button1 = wx.Button(self, -1, '+', (10, 10))
        button2 = wx.Button(self, -1, '-', (10, 60))

        self.Bind(wx.EVT_BUTTON, self.OnPlus, id=button1.GetId())
        self.Bind(wx.EVT_BUTTON, self.OnMinus, id=button2.GetId())

    def OnPlus(self, event):
        value = int(self.text.GetLabel())
        value = value + 1
        self.text.SetLabel(str(value))

    def OnMinus(self, event):
        value = int(self.text.GetLabel())
        value = value - 1
```

```
        self.text.SetLabel(str(value))


class RightPanel(wx.Panel):
    def __init__(self, parent, id):
        wx.Panel.__init__(self, parent, id, style=wx.BORDER_SUNKEN)
        self.text = wx.StaticText(self, -1, '0', (40, 60))


class Communicate(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(280, 200))

        panel = wx.Panel(self, -1)
        self.rightPanel = RightPanel(panel, -1)

        leftPanel = LeftPanel(panel, -1)

        hbox = wx.BoxSizer()
        hbox.Add(leftPanel, 1, wx.EXPAND | wx.ALL, 5)
        hbox.Add(self.rightPanel, 1, wx.EXPAND | wx.ALL, 5)

        panel.SetSizer(hbox)
        self.Centre()
        self.Show(True)

app = wx.App()
Communicate(None, -1, 'widgets communicate')
app.MainLoop()
```

In our example we have two panels. A left and right panel. The left panel has two buttons. The right panel has one static text. The buttons change the number displayed in the static text. The question is, how do we grab the reference to the static text?

If all the widgets are within one class, it is trivial. But what if those widgets are created in different classes? In such situations, we must get the reference via the hierarchy.

```
panel = wx.Panel(self, -1)
self.rightPanel = RightPanel(panel, -1)


leftPanel = LeftPanel(panel, -1)
```

Notice that the right panel **must** be defined before the left panel. It is because during the construction of the left panel, we are looking for the static text widget, which is defined in

the right panel. Logically, we cannot get reference to non existing widget.

```
self.text = parent.GetParent().rightPanel.text
```

The answer is here. Each widget has a parent argument. In our example, parent is a panel on which we display both left and right panels. By calling parent.GetParent() we get reference to the frame widget. The frame widget has reference to the rightPanel. Finally, right panel has reference to the static text widget.

Figure: how widgets communicate

Home ǂ Contents ǂ Top of Page

Menus and Toolbars in wxPython

Creating a MenuBar

A menubar is one of the most visible parts of the GUI application. It is a group of commands located in various menus. While in console applications you had to remember all those arcane commands, here we have most of the commands grouped into logical parts. There are accepted standards that further reduce the amount of time spending to learn a new application. To implement a menubar in wxPython we need to have three things. A *wx.MenuBar*, a *wx.Menu* and a *wx.MenuItem*.

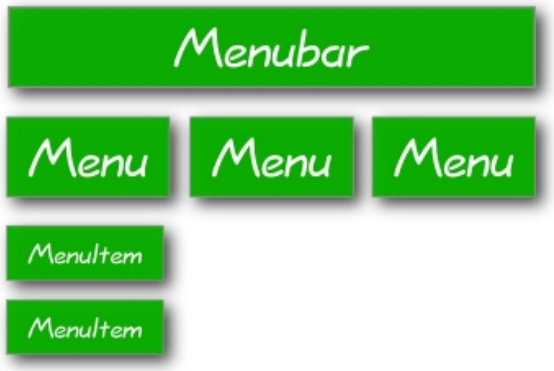


Figure: A MenuBar architecture

A Simple menu example

Creating a menubar in wxPython is very simple. Just a few lines of code.

```
#!/usr/bin/python

# simplemenu.py

import wx

class SimpleMenu(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        menubar = wx.MenuBar()
        file = wx.Menu()
        file.Append(-1, 'Quit', 'Quit application')
        menubar.Append(file, '&File')
        self.SetMenuBar(menubar)

        self.Centre()
```

```
        self.Show(True)

app = wx.App()
SimpleMenu(None, -1, 'simple menu example')
app.MainLoop()
```

```
menubar = wx.MenuBar()
```

First we create a menubar object.

```
file = wx.Menu()
```

Next we create a menu object.

```
file.Append(-1, 'Quit', 'Quit application')
```

We append a menu item into the menu object. The first parameter is the id of the menu item. The second parameter is the name of the menu item. The last parameter defines the short helpstring that is displayed on the statusbar, when the menu item is selected. Here we did not create a *wx.MenuItem* explicitely. It was created by the *Append()* method behind the scenes. Later on, we will create a *wx.MenuItem* manually.

```
menubar.Append(file, '&File')
self.SetMenuBar(menubar)
```

After that, we append a menu into the menubar. The & character creates an accelerator key. The character that follows the & is underlined. This way the menu is accessible via the alt + F shortcut. In the end, we call the *SetMenuBar()* method. This method belongs to the wx.Frame widget. It sets up the menubar.

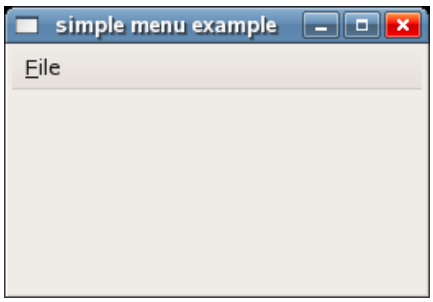


Figure: A simple menu example

A dockable menubar

Under Linux, we can create a dockable menubar. This feature is not commoly seen in applications. But similar thing can be seen on Mac OS. Mac users do not have a menubar in the toplevet application window. The menubar is implemented outside the main window.

```
#!/usr/bin/python

# dockable.py

import wx

class Dockable(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        menubar = wx.MenuBar(wx.MB_DOCKABLE)
        file = wx.Menu()
        edit = wx.Menu()
        view = wx.Menu()
        insr = wx.Menu()
        form = wx.Menu()
        tool = wx.Menu()
        help = wx.Menu()

        menubar.Append(file, '&File')
        menubar.Append(edit, '&Edit')
        menubar.Append(view, '&View')
        menubar.Append(insr, '&Insert')
        menubar.Append(form, '&Format')
        menubar.Append(tool, '&Tools')
        menubar.Append(help, '&Help')
        self.SetMenuBar(menubar)

        self.Centre()
        self.Show(True)

app = wx.App()
Dockable(None, -1, 'Dockable menubar')
app.MainLoop()
```

```
menubar = wx.MenuBar(wx.MB_DOCKABLE)
```

We create a dockable menubar by providing a wx.MB_DOCKABLE flag to the constructor.

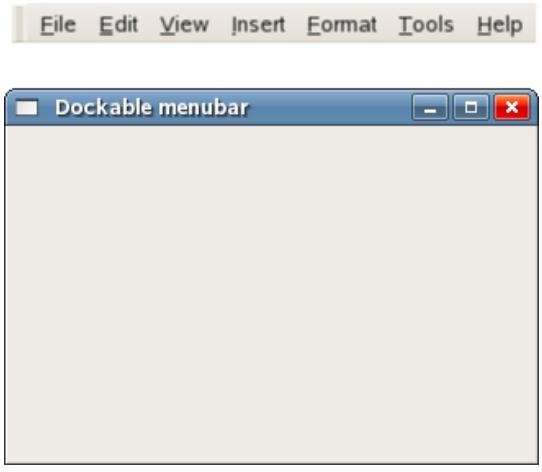


Figure: A dockable menubar

Icons, shortcuts, events

In the next section we will further enhance our menu example. We will see, how we can add icons to our menus. Icons make our applications more visually attractive. Further, they help us understand the menu commands. We will see, how we can add shortcuts to our menus. Shortcuts are not a relict from the past. They enable us to work more quickly with our applications. One of the most widely used shortcut is the Ctrl + S one. There are not many people, that would not know the meaning of this shortcut. It is more handy to press this shortcut, than to move a mouse pointer to the menubar, click a File menu and select the Save commnand. Shortcuts are a productivity boost to most users. We will also briely touch events.

```
#!/usr/bin/python

# menuexample.py

import wx

class MenuExample(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        menubar = wx.MenuBar()
        file = wx.Menu()
        quit = wx.MenuItem(file, 1, '&Quit\tCtrl+Q')
        quit.SetBitmap(wx.Bitmap('icons/exit.png'))
        file.AppendItem(quit)

        self.Bind(wx.EVT_MENU, self.OnQuit, id=1)

        menubar.Append(file, '&File')
        self.SetMenuBar(menubar)

        self.Centre()
        self.Show(True)

    def OnQuit(self, event):
        self.Close()

app = wx.App()
MenuExample(None, -1, '')
app.MainLoop()
```

```
quit = wx.MenuItem(file, 1, '&Quit\tCtrl+Q')
quit.SetBitmap(wx.Bitmap('icons/exit.png'))
file.AppendItem(quit)
```

If we want to add shortcuts and icons to our menus, we have to manually create a *wx.MenuItem*. So far we have created menuitems indirectly. The & character specifies an accelerator key. The following character is underlined. The actual shortcut is defined by the combination of characters. We have specified Ctrl + Q characters. So if we press Ctrl + Q, we close the application. We put a tab character between the &

character and the shortcut. This way, we manage to put some space between them. To provide an icon for a menuitem, we call a *SetBitmap()* method. A manually created menuitem is appended to the menu by calling the *AppendItem()* method.

```
self.Bind(wx.EVT_MENU, self.OnQuit, id=1)
```

If we select a quit menu item or press a keyboard shortcut, a *wx.EVT_MENU* event is generated. We bind an event handler to the event. The event handler is a method, that is being called. In our example, the *OnQuit()* method closes the application. There can be several menuitems, so we have to give a unique id to each of them. Working with events is very easy and straightforward in wxPython. We will talk about events in a separate chapter.

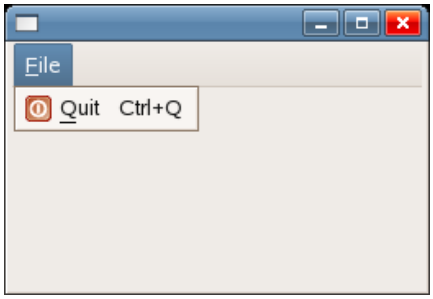


Figure: A menu example

Submenus

Each menu can also have a submenu. This way we can group similar commnads into groups. For example we can place commands that hide/show various toolbars like personal bar, address bar, status bar or navigation bar into a submenu called toolbars. Within a menu, we can seperate commands with a separator. It is a simple line. It is common practice to separate commands like new, open, save from commands like print, print preview with a single separator. In our example we will see, how we can create submenus and menu separators.

```
#!/usr/bin/python

# submenu.py

import wx

ID_QUIT = 1

class SubmenuExample(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 250))

        menubar = wx.MenuBar()

        file = wx.Menu()
```

```
        file.Append(-1, '&New')
        file.Append(-1, '&Open')
        file.Append(-1, '&Save')
        file.AppendSeparator()

        imp = wx.Menu()
        imp.Append(-1, 'Import newsfeed list...')
        imp.Append(-1, 'Import bookmarks...')
        imp.Append(-1, 'Import mail...')

        file.AppendMenu(-1, 'I&mport', imp)

        quit = wx.MenuItem(file, ID_QUIT, '&Quit\tCtrl+W')
        quit.SetBitmap(wx.Bitmap('icons/exit.png'))
        file.AppendItem(quit)

        self.Bind(wx.EVT_MENU, self.OnQuit, id=ID_QUIT)

        menubar.Append(file, '&File')
        self.SetMenuBar(menubar)

        self.Centre()
        self.Show(True)

    def OnQuit(self, event):
        self.Close()

app = wx.App()
SubmenuExample(None, -1, 'Submenu')
app.MainLoop()
```

```
file.AppendSeparator()
```

A menu separator is appended with the AppendSeparator() method.

```
imp = wx.Menu()
imp.Append(-1, 'Import newsfeed list...')
imp.Append(-1, 'Import bookmarks...')
imp.Append(-1, 'Import mail...')

file.AppendMenu(-1, 'I&mport', imp)
```

Creating a submenu is trivial. First, we create a menu. Then we append menu items. A submenu is created by calling the *AppenMenu()* on the menu object.

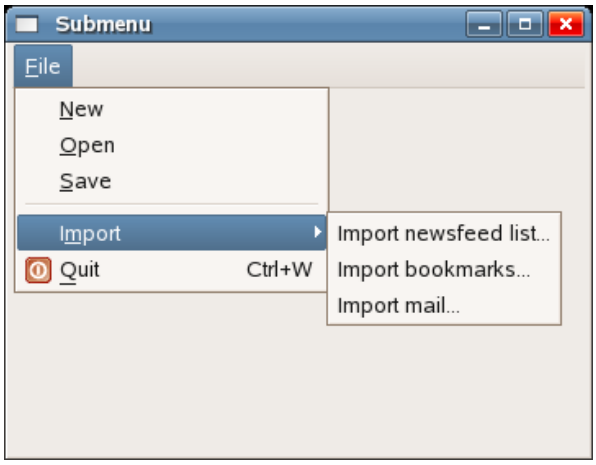


Figure: A submenu example

Various menu items

There are tree kinds of menu items.

- normal item
- check item
- radio item

```
#!/usr/bin/python

# checkmenuitem.py

import wx

ID_STAT = 1
ID_TOOL = 2

class CheckMenuItem(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 250))

        menubar = wx.MenuBar()
        file = wx.Menu()
        view = wx.Menu()
        self.shst = view.Append(ID_STAT, 'Show statubar', 'Show Statusbar', kind=
        self.shtl = view.Append(ID_TOOL, 'Show toolbar', 'Show Toolbar', kind=wx.
        view.Check(ID_STAT, True)
        view.Check(ID_TOOL, True)

        self.Bind(wx.EVT_MENU, self.ToggleStatusBar, id=ID_STAT)
        self.Bind(wx.EVT_MENU, self.ToggleToolBar, id=ID_TOOL)

        menubar.Append(file, '&File')
        menubar.Append(view, '&View')
        self.SetMenuBar(menubar)

        self.toolbar = self.CreateToolBar()
        self.toolbar.AddLabelTool(3, '', wx.Bitmap('icons/quit.png'))

        self.statusbar = self.CreateStatusBar()
        self.Centre()
        self.Show(True)
```

```
    def ToggleStatusBar(self, event):
        if self.shst.IsChecked():
            self.statusbar.Show()
        else:
            self.statusbar.Hide()

    def ToggleToolBar(self, event):
        if self.shtl.IsChecked():
            self.toolbar.Show()
        else:
            self.toolbar.Hide()

app = wx.App()
CheckMenuItem(None, -1, 'check menu item')
app.MainLoop()
```

```
self.shst = view.Append(ID_STAT, 'Show statubar', 'Show Statusbar', kind=wx.ITEM_
self.shtl = view.Append(ID_TOOL, 'Show toolbar', 'Show Toolbar', kind=wx.ITEM_CHE
```

If we want to append a check menu item, we set a *kind* parameter to *wx.ITEM_CHECK*. The default parameter is *wx.ITEM_NORMAL*. The *Append()* method returns a *wx.MenuItem*.

```
view.Check(ID_STAT, True)
view.Check(ID_TOOL, True)
```

When the application starts, both statusbar and toolbar are visible. So we check both menu items with the *Check()* method.

```
def ToggleStatusBar(self, event):
    if self.shst.IsChecked():
        self.statusbar.Show()
    else:
        self.statusbar.Hide()
```

We show or hide the statusbar according to the state of the check menu item. We find out the state of the check menu item with the *IsChecked()* method. Same with toolbar.

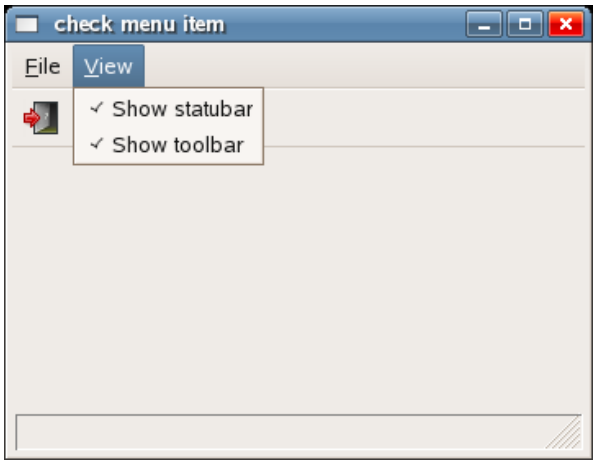


Figure: check menu item

Context menu

It is a list of commands that appears under some context. For example, in a Firefox web browser, when we right click on a web page, we get a context menu. Here we can reload a page, go back or view page source. If we right click on a toolbar, we get another context menu for managing toolbars. Context menus are sometimes called popup menus.

```
#!/usr/bin/python

# contextmenu.py

import wx


class MyPopupMenu(wx.Menu):
    def __init__(self, parent):
        wx.Menu.__init__(self)

        self.parent = parent

        minimize = wx.MenuItem(self, wx.NewId(), 'Minimize')
        self.AppendItem(minimize)
        self.Bind(wx.EVT_MENU, self.OnMinimize, id=minimize.GetId())

        close = wx.MenuItem(self, wx.NewId(), 'Close')
        self.AppendItem(close)
        self.Bind(wx.EVT_MENU, self.OnClose, id=close.GetId())


    def OnMinimize(self, event):
        self.parent.Iconize()

    def OnClose(self, event):
        self.parent.Close()


class ContextMenu(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRightDown)

        self.Center()
        self.Show()

    def OnRightDown(self, event):
        self.PopupMenu(MyPopupMenu(self), event.GetPosition())


app = wx.App()
frame = ContextMenu(None, -1, 'context menu')
app.MainLoop()
```

```
class MyPopupMenu(wx.Menu):
    def __init__(self, parent):
```

```
        wx.Menu.__init__(self)
```

We create a separate *wx.Menu* class. Here we define two commands. Close and minimize window.

```
self.Bind(wx.EVT_RIGHT_DOWN, self.OnRightDown)
```

If we right click on the frame, we call the *OnRightDown()* method. For this, we use the *wx.EVT_RIGHT_DOWN* event binder.

```
def OnRightDown(self, event):
    self.PopupMenu(MyPopupMenu(self), event.GetPosition())
```

In the OnRightDown() method, we call the *PopupMenu()* method. This method shows the context menu. The first parameter is the menu to be shown. The second parameter is the position, where the context menu appears. The context menus appear at the point of the mouse cursor. To get the actual mouse position, we call the *GetPosition()* menthod.

Toolbars

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

```
CreateToolBar(long style=-1, int winid=-1, String name=ToolBarNameStr)
```

To create a toolbar, we call the *CreateToolBar()* method of the frame widget.

```
#!/usr/bin/python

# simpletoolbar.py

import wx

class SimpleToolbar(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 200))

        toolbar = self.CreateToolBar()
        toolbar.AddLabelTool(wx.ID_EXIT, '', wx.Bitmap('../icons/exit.png'))
        toolbar.Realize()

        self.Bind(wx.EVT_TOOL, self.OnExit, id=wx.ID_EXIT)

        self.Centre()
        self.Show(True)

    def OnExit(self, event):
        self.Close()


app = wx.App()
SimpleToolbar(None, -1, 'simple toolbar')
```

```
app.MainLoop()
```

```
toolbar.AddLabelTool(wx.ID_EXIT, '', wx.Bitmap('../icons/exit.png'))
```

To create a toolbar button, we call the *AddLabelTool()* method.

```
toolbar.Realize()
```

After we have put our items to the toolbar, we call the *Realize()* method. Calling this method is not obligatory on Linux. On windows it is.

```
self.Bind(wx.EVT_TOOL, self.OnExit, id=wx.ID_EXIT)
```

To handle toolbar events, we use the *wx.EVT_TOOL* event binder.

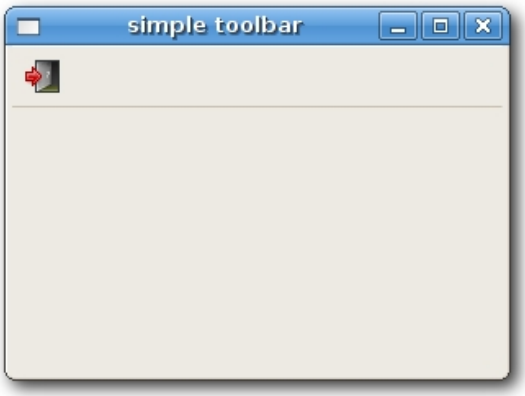


Figure: simple toolbar

If we want to create more than one toolbars, we must do it differently.

```
#!/usr/bin/python

# toolbars.py

import wx

class Toolbars(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 200))

        vbox = wx.BoxSizer(wx.VERTICAL)

        toolbar1 = wx.ToolBar(self, -1)
        toolbar1.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/new.png'))
        toolbar1.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/open.png'))
        toolbar1.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/save.png'))
        toolbar1.Realize()

        toolbar2 = wx.ToolBar(self, -1)
        toolbar2.AddLabelTool(wx.ID_EXIT, '', wx.Bitmap('../icons/exit.png'))
        toolbar2.Realize()
```

```
        vbox.Add(toolbar1, 0, wx.EXPAND)
        vbox.Add(toolbar2, 0, wx.EXPAND)

        self.Bind(wx.EVT_TOOL, self.OnExit, id=wx.ID_EXIT)

        self.SetSizer(vbox)
        self.Centre()
        self.Show(True)

    def OnExit(self, event):
        self.Close()


app = wx.App()
Toolbars(None, -1, 'toolbars')
app.MainLoop()
```

```
toolbar1 = wx.ToolBar(self, -1)
...
toolbar2 = wx.ToolBar(self, -1)
```

We create two toolbar objects. And put them into a vertical box.

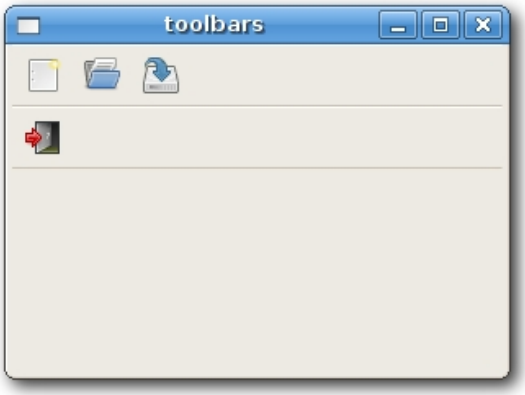


Figure: toolbars

Sometimes we need to create a vertical toolbar. Vertical toolbars are often seen in graphics applications like Inkscape or Xara Xtreme.

```
#!/usr/bin/python

# verticaltoolbar.py

import wx

class VerticalToolbar(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(240, 300))

        toolbar = self.CreateToolBar(wx.TB_VERTICAL)
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/select.gif'))
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/freehand.gif'))
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/shapeed.gif'))
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/pen.gif'))
```

```
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/rectangle.gif'))
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/ellipse.gif'))
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/qs.gif'))
        toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/text.gif'))

        toolbar.Realize()


        self.Centre()
        self.Show(True)

    def OnExit(self, event):
        self.Close()


app = wx.App()
VerticalToolbar(None, -1, 'vertical toolbar')
app.MainLoop()
```

```
toolbar = self.CreateToolBar(wx.TB_VERTICAL)
```

Here we create a vertical toolbar.

```
toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/select.gif'))
toolbar.AddLabelTool(wx.ID_ANY, '', wx.Bitmap('../icons/freehand.gif'))
...
```

I have borrowed icons from the **Xara Xtreme** graphics application.

Figure: vertical toolbar

In the following example, we will show, how we can enable and disable toolbar buttons. We will also see a separator line.

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

```
# enabledisable.py

import wx

class EnableDisable(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.count = 5

        self.toolbar = self.CreateToolBar()
        self.toolbar.AddLabelTool(wx.ID_UNDO, '', wx.Bitmap('../icons/undo.png'))
        self.toolbar.AddLabelTool(wx.ID_REDO, '', wx.Bitmap('../icons/redo.png'))
        self.toolbar.EnableTool(wx.ID_REDO, False)
        self.toolbar.AddSeparator()
        self.toolbar.AddLabelTool(wx.ID_EXIT, '', wx.Bitmap('../icons/exit.png'))
        self.toolbar.Realize()

        self.Bind(wx.EVT_TOOL, self.OnExit, id=wx.ID_EXIT)
        self.Bind(wx.EVT_TOOL, self.OnUndo, id=wx.ID_UNDO)
        self.Bind(wx.EVT_TOOL, self.OnRedo, id=wx.ID_REDO)

        self.Centre()
        self.Show(True)

    def OnUndo(self, event):
        if self.count > 1 and self.count <= 5:
            self.count = self.count - 1

        if self.count == 1:
            self.toolbar.EnableTool(wx.ID_UNDO, False)

        if self.count == 4:
            self.toolbar.EnableTool(wx.ID_REDO, True)

    def OnRedo(self, event):
        if self.count < 5 and self.count >= 1:
            self.count = self.count + 1

        if self.count == 5:
            self.toolbar.EnableTool(wx.ID_REDO, False)

        if self.count == 2:
            self.toolbar.EnableTool(wx.ID_UNDO, True)

    def OnExit(self, event):
        self.Close()

app = wx.App()
EnableDisable(None, -1, 'enable disable')
app.MainLoop()
```

In our example, we have three toolbar buttons. One button is for exiting the application. The other two buttons are undo and redo buttons. They simulate undo/redo functionality in an application. (for a real example, see tips and tricks) We have 4 changes. The undo and redo butons are disabled accordingly.

```
self.toolbar.EnableTool(wx.ID_REDO, False)
self.toolbar.AddSeparator()
```

In the beginning, the redo button is disabled. We do it by calling the

EnableTool() method. We can create some logical groups within a toolbar. We can separate various groups of buttons by a small vertical line. To do this, we call the *AddSeparator()* method.

```
def OnUndo(self, event):
    if self.count > 1 and self.count <= 5:
        self.count = self.count - 1

    if self.count == 1:
        self.toolbar.EnableTool(wx.ID_UNDO, False)

    if self.count == 4:
        self.toolbar.EnableTool(wx.ID_REDO, True)
```

We simulate undo and redo functionality. We have 4 changes. If there is nothing left to undo, the undo button is disabled. After undoing the first change, we enable the redo button. Same logic applies for the *OnRedo()* method.

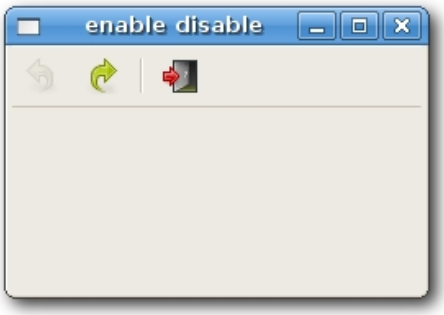


Figure: enable disable buttons

Home ǂ Contents ǂ Top of Page

Layout management in wxPython

A typical application consists of various widgets. Those widgets are placed inside container widgets. A programmer must manage the layout of the application. This is not an easy task. In wxPython we have two options.

- absolute positioning
- sizers

Absolute Positioning

The programmer specifies the position and the size of each widget in pixels. When you use absolute positioning, you have to understand several things.

- the size and the position of a widget do not change, if you resize a window
- applications look different (crappy) on various platforms
- changing fonts in your application might spoil the layout
- if you decide to change your layout, you must completely redo your layout, which is tedious and time consuming

There might be situations, where we can possibly use absolute positioning. For example, my tutorials. I do not want to make the examples too difficult, so I often use absolute positioning to explain a topic. But mostly, in real world programs, programmers use sizers.

In our example we have a simple skeleton of a text editor. If we resize the window, the size of out wx.TextCtrl does not change as we would expect.

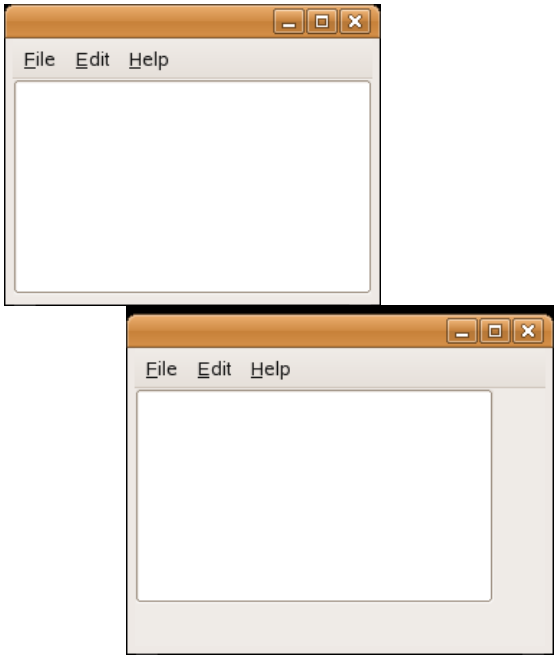


Figure: before resizement

Figure: after resizement

```
#!/usr/bin/python

# absolute.py

import wx

class Absolute(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 180))
        panel = wx.Panel(self, -1)

        menubar = wx.MenuBar()
        file = wx.Menu()
        edit = wx.Menu()
        help = wx.Menu()

        menubar.Append(file, '&File')
        menubar.Append(edit, '&Edit')
        menubar.Append(help, '&Help')
        self.SetMenuBar(menubar)

        wx.TextCtrl(panel, -1, pos=(-1, -1), size=(250, 150))

        self.Centre()
        self.Show(True)

app = wx.App(0)
Absolute(None, -1, '')
app.MainLoop()
```

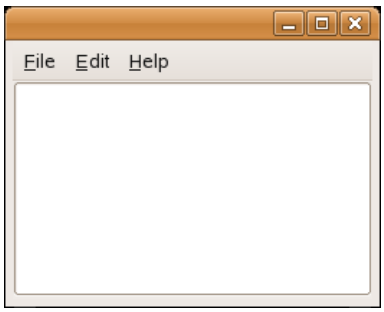
```
wx.TextCtrl(panel, -1, pos=(-1, -1), size=(250, 150))
```

We do the absolute positioning in the constructor of the wx.TextCtrl. In our case, we provide the default position for the widget. The width is 250px and the height 150px.

Using sizers

Sizers do address all those issues, we mentioned by absolute positioning. We can choose among these sizers.

- wx.BoxSizer
- wx.StaticBoxSizer
- wx.GridSizer
- wx.FlexGridSizer
- wx.GridBagSizer



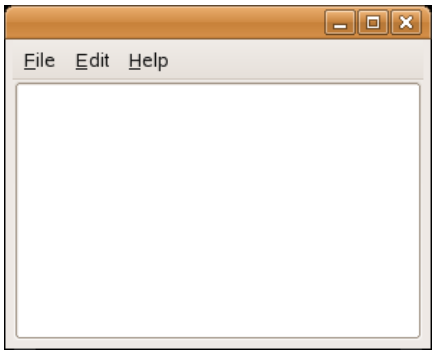


Figure: before resizement

Figure: after resizement

```
#!/usr/bin/python

# sizer.py

import wx

class Sizer(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 180))

        menubar = wx.MenuBar()
        file = wx.Menu()
        edit = wx.Menu()
        help = wx.Menu()

        menubar.Append(file, '&File')
        menubar.Append(edit, '&Edit')
        menubar.Append(help, '&Help')
        self.SetMenuBar(menubar)

        wx.TextCtrl(self, -1)

        self.Centre()
        self.Show(True)

app = wx.App(0)
Sizer(None, -1, '')
app.MainLoop()
```

Ok, so you are saying that you don't see any sizers in the example? Well, the code example was a bit tricky. Actually, we placed the wx.TextCtrl inside the wx.Frame widget. The wx.Frame widget has a special built-in sizer. We can put only one widget inside the wx.Frame container. The child widget occupies all the space, which is not given to the borders, menu, toolbar and the statusbar.

wx.BoxSizer

This sizer enables us to put several widgets into a row or a column. We can put another sizer into an existing sizer. This way we can create very complex layouts.

```
box = wx.BoxSizer(integer orient)
box.Add(wx.Window window, integer proportion=0, integer flag = 0, integer border = 0)
```

The orientation can be wx.VERTICAL or wx.HORIZONTAL. Adding widgets into the wx.BoxSizer is done via the Add() method. In order to understand it, we need to look at its parameters.

The proportion parameter defines the ratio of how will the widgets change in the defined orientation. Let's assume we have tree buttons with the proportions 0, 1, and

2. They are added into a horizontal wx.BoxSizer. Button with proportion 0 will not change at all. Button with proportion 2 will change twice more than the one with proportion 1 in the horizontal dimension.

With the flag parameter you can further configure the behaviour of the widgets within a wx.BoxSizer. We can control the border between the widgets. We add some space between widgets in pixels. In order to apply border we need to define sides, where the border will be used. We can combine them with the | operator. e.g wx.LEFT | wx.BOTTOM. We can choose between these flags:

- wx.LEFT
- wx.RIGHT
- wx.BOTTOM
- wx.TOP
- wx.ALL

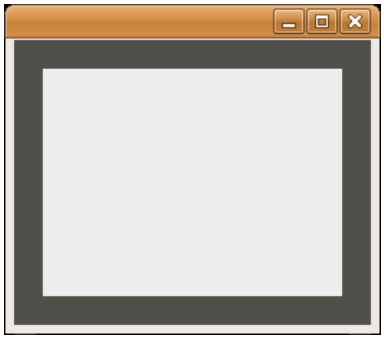


Figure: border around a panel

```
#!/usr/bin/python

# border.py

import wx

class Border(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 200))

        panel = wx.Panel(self, -1)
        panel.SetBackgroundColour('#4f5049')
        vbox = wx.BoxSizer(wx.VERTICAL)

        midPan = wx.Panel(panel, -1)
        midPan.SetBackgroundColour('#ededed')

        vbox.Add(midPan, 1, wx.EXPAND | wx.ALL, 20)
        panel.SetSizer(vbox)
        self.Centre()
        self.Show(True)

app = wx.App()
Border(None, -1, '')
app.MainLoop()
```

```
vbox.Add(midPan, 1, wx.EXPAND | wx.ALL, 20)
```

In border.py we have placed a 20 px border around a midPan panel. wx.ALL applies the border size to all four sides.

If we use wx.EXPAND flag, our widget will use all the space that has been allotted to it. Lastly, we can also define the alignment of our widgets. We do it with the following flags :

- wx.ALIGN_LEFT
- wx.ALIGN_RIGHT
- wx.ALIGN_TOP
- wx.ALIGN_BOTTOM
- wx.ALIGN_CENTER_VERTICAL
- wx.ALIGN_CENTER_HORIZONTAL
- wx.ALIGN_CENTER

Go To Class

In the following example we introduce several important ideas.

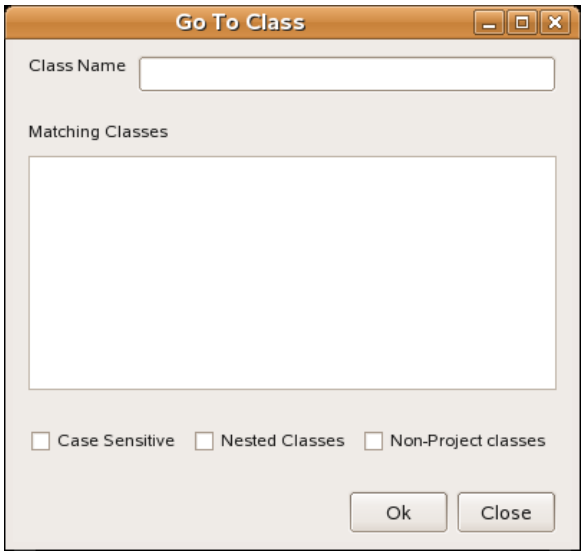


Figure: A go to class window

```
#!/usr/bin/python

# gotoclass.py

import wx

class GoToClass(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(390, 350))
        panel = wx.Panel(self, -1)

        font = wx.SystemSettings_GetFont(wx.SYS_SYSTEM_FONT)
        font.SetPointSize(9)

        vbox = wx.BoxSizer(wx.VERTICAL)

        hbox1 = wx.BoxSizer(wx.HORIZONTAL)
        st1 = wx.StaticText(panel, -1, 'Class Name')
        st1.SetFont(font)
        hbox1.Add(st1, 0, wx.RIGHT, 8)
        tc = wx.TextCtrl(panel, -1)
        hbox1.Add(tc, 1)
        vbox.Add(hbox1, 0, wx.EXPAND | wx.LEFT | wx.RIGHT | wx.TOP, 10)

        vbox.Add((-1, 10))

        hbox2 = wx.BoxSizer(wx.HORIZONTAL)
        st2 = wx.StaticText(panel, -1, 'Matching Classes')
        st2.SetFont(font)
        hbox2.Add(st2, 0)
        vbox.Add(hbox2, 0, wx.LEFT | wx.TOP, 10)

        vbox.Add((-1, 10))
```

```
        hbox3 = wx.BoxSizer(wx.HORIZONTAL)
        tc2 = wx.TextCtrl(panel, -1, style=wx.TE_MULTILINE)
        hbox3.Add(tc2, 1, wx.EXPAND)
        vbox.Add(hbox3, 1, wx.LEFT | wx.RIGHT | wx.EXPAND, 10)

        vbox.Add((-1, 25))

        hbox4 = wx.BoxSizer(wx.HORIZONTAL)
        cb1 = wx.CheckBox(panel, -1, 'Case Sensitive')
        cb1.SetFont(font)
        hbox4.Add(cb1)
        cb2 = wx.CheckBox(panel, -1, 'Nested Classes')
        cb2.SetFont(font)
        hbox4.Add(cb2, 0, wx.LEFT, 10)
        cb3 = wx.CheckBox(panel, -1, 'Non-Project classes')
        cb3.SetFont(font)
        hbox4.Add(cb3, 0, wx.LEFT, 10)
        vbox.Add(hbox4, 0, wx.LEFT, 10)

        vbox.Add((-1, 25))

        hbox5 = wx.BoxSizer(wx.HORIZONTAL)
        btn1 = wx.Button(panel, -1, 'Ok', size=(70, 30))
        hbox5.Add(btn1, 0)
        btn2 = wx.Button(panel, -1, 'Close', size=(70, 30))
        hbox5.Add(btn2, 0, wx.LEFT | wx.BOTTOM , 5)
        vbox.Add(hbox5, 0, wx.ALIGN_RIGHT | wx.RIGHT, 10)

        panel.SetSizer(vbox)
        self.Centre()
        self.Show(True)


app = wx.App()
GoToClass(None, -1, 'Go To Class')
app.MainLoop()
```

The layout is straitforward. We create one vertical sizer. We put then five horizontal sizers into it.

```
font = wx.SystemSettings_GetFont(wx.SYS_SYSTEM_FONT)
font.SetPointSize(9)
```

The default system font was 10px. On my platform, it was too big for this kind of window. So I set the font size to 9px.

```
vbox.Add(hbox3, 1, wx.LEFT | wx.RIGHT | wx.EXPAND, 10)

vbox.Add((-1, 25))
```

We already know that we can control the distance among widgets by combining the flag parameter with the border parameter. But there is one real constraint. In the Add() method we can specify only one border for all given sides. In our example, we give 10px to the right and to the left. But we cannot give 25 px to the bottom. What we can do is to give 10px to the bottom, or 0px. If we omit wx.BOTTOM. So if we need different values, we can add some extra space. With the Add() method, we can insert widgets and space as well.

```
vbox.Add(hbox5, 0, wx.ALIGN_RIGHT | wx.RIGHT, 10)
```

We place the two buttons on the right side of the window. How do we do it? Three things are important to achieve this. The proportion, the align flag and the wx.EXPAND flag. The proportion must be zero. The buttons should not change their size, when we resize our window. We must not specify wx.EXPAND flag. The buttons

occopy only the area that has been alotted to it. And finally, we must specify the wx.ALIGN_RIGHT flag. The horizontal sizer spreads from the left side of the window to the right side. So if we specify wx.ALIGN_RIGHT flag, the buttons are placed to the right side. Exactly, as we wanted.

Find/Replace Dialog

A complex example follows. We will create a find/replace dialog. This kind of dialog can be found in the Eclipse IDE.

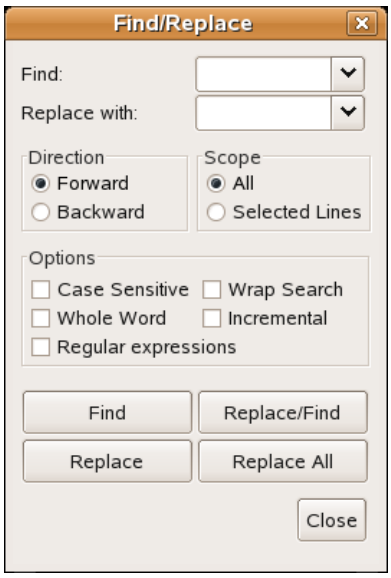


Figure: A complex example using wx.BoxSizer

```
#!/usr/bin/python

# Find/Replace Dialog

import wx

class FindReplace(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(255, 365))

        vbox_top = wx.BoxSizer(wx.VERTICAL)
        panel = wx.Panel(self, -1)

        vbox = wx.BoxSizer(wx.VERTICAL)

        # panel1

        panel1 = wx.Panel(panel, -1)
        grid1 = wx.GridSizer(2, 2)
        grid1.Add(wx.StaticText(panel1, -1, 'Find: ', (5, 5)), 0,  wx.ALIGN_CENTER_VERTICAL)
        grid1.Add(wx.ComboBox(panel1, -1, size=(120, -1)))
        grid1.Add(wx.StaticText(panel1, -1, 'Replace with: ', (5, 5)), 0, wx.ALIGN_CENTER_VERTIC
        grid1.Add(wx.ComboBox(panel1, -1, size=(120, -1)))

        panel1.SetSizer(grid1)
        vbox.Add(panel1, 0, wx.BOTTOM | wx.TOP, 9)

        # panel2

        panel2 = wx.Panel(panel, -1)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)

        sizer21 = wx.StaticBoxSizer(wx.StaticBox(panel2, -1, 'Direction'), orient=wx.VERTICAL)
        sizer21.Add(wx.RadioButton(panel2, -1, 'Forward', style=wx.RB_GROUP))
        sizer21.Add(wx.RadioButton(panel2, -1, 'Backward'))
```

```
        hbox2.Add(sizer21, 1, wx.RIGHT, 5)

        sizer22 = wx.StaticBoxSizer(wx.StaticBox(panel2, -1, 'Scope'), orient=wx.VERTICAL)
        # we must define wx.RB_GROUP style, otherwise all 4 RadioButtons would be mutually exclu
        sizer22.Add(wx.RadioButton(panel2, -1, 'All', style=wx.RB_GROUP))
        sizer22.Add(wx.RadioButton(panel2, -1, 'Selected Lines'))
        hbox2.Add(sizer22, 1)

        panel2.SetSizer(hbox2)
        vbox.Add(panel2, 0, wx.BOTTOM, 9)

        # panel3

        panel3 = wx.Panel(panel, -1)
        sizer3 = wx.StaticBoxSizer(wx.StaticBox(panel3, -1, 'Options'), orient=wx.VERTICAL)
        vbox3 = wx.BoxSizer(wx.VERTICAL)
        grid = wx.GridSizer(3, 2, 0, 5)
        grid.Add(wx.CheckBox(panel3, -1, 'Case Sensitive'))
        grid.Add(wx.CheckBox(panel3, -1, 'Wrap Search'))
        grid.Add(wx.CheckBox(panel3, -1, 'Whole Word'))
        grid.Add(wx.CheckBox(panel3, -1, 'Incremental'))
        vbox3.Add(grid)
        vbox3.Add(wx.CheckBox(panel3, -1, 'Regular expressions'))
        sizer3.Add(vbox3, 0, wx.TOP, 4)

        panel3.SetSizer(sizer3)
        vbox.Add(panel3, 0, wx.BOTTOM, 15)

        # panel4

        panel4 = wx.Panel(panel, -1)
        sizer4 = wx.GridSizer(2, 2, 2, 2)
        sizer4.Add(wx.Button(panel4, -1, 'Find', size=(120, -1)))
        sizer4.Add(wx.Button(panel4, -1, 'Replace/Find', size=(120, -1)))
        sizer4.Add(wx.Button(panel4, -1, 'Replace', size=(120, -1)))
        sizer4.Add(wx.Button(panel4, -1, 'Replace All', size=(120, -1)))

        panel4.SetSizer(sizer4)
        vbox.Add(panel4, 0, wx.BOTTOM, 9)

        # panel5

        panel5 = wx.Panel(panel, -1)
        sizer5 = wx.BoxSizer(wx.HORIZONTAL)
        sizer5.Add((191, -1), 1, wx.EXPAND | wx.ALIGN_RIGHT)
        sizer5.Add(wx.Button(panel5, -1, 'Close', size=(50, -1)))

        panel5.SetSizer(sizer5)
        vbox.Add(panel5, 1, wx.BOTTOM, 9)

        vbox_top.Add(vbox, 1, wx.LEFT, 5)
        panel.SetSizer(vbox_top)

        self.Centre()
        self.ShowModal()
        self.Destroy()


app = wx.App()
FindReplace(None, -1, 'Find/Replace')
app.MainLoop()
```

(Remark for Windows users, put self.SetClientSize(panel.GetBestSize()) line before the ShowModal() method.) Before we actually code our layout, we should have an idea of how we are going to achieve our goal. A simple sketch of a more complex dialog or window might be handy. If we look at the dialog screenshot, we clearly see, that we can divite it into five parts. The close button will also have a separate panel. Each part will be a unique wx.Panel. Together we have then 6 panels. The first panel is a parent panel. It will host all the five panels, that we have identified.

All five panels reside in one column. So the parent panel will have a vertical

wx.BoxSizer. Apart from wx.BoxSizer-s we use wx.GridSizer-s as well. The wx.GridSizer will be explained in the next section. Well, there is not much to explain, since the usage of the wx.GridSizer is pretty straightforward.

```
sizer4 = wx.GridSizer(2, 2, 2, 2)
sizer4.Add(wx.Button(panel4, -1, 'Find', size=(120, -1)))
sizer4.Add(wx.Button(panel4, -1, 'Replace/Find', size=(120, -1)))
sizer4.Add(wx.Button(panel4, -1, 'Replace', size=(120, -1)))
sizer4.Add(wx.Button(panel4, -1, 'Replace All', size=(120, -1)))
```

In our example the wx.GridSizer is very useful. We need four buttons of the same size in a particular panel. It is a job for wx.GridSizer, since it organizes all widgets in a grid of cells. Those cells all have the same size and the same width.

wx.GridSizer

wx.GridSizer lays out widgets in two dimensional table. Each cell within the table has the same size.

```
wx.GridSizer(int rows=1, int cols=0, int vgap=0, int hgap=0)
```

In the constructor we specify the number of rows and columns in the table. And the vertical and horizontal space between our cells.

In our example we create a skeleton of a calculator. It is a perfect example for wx.GridSizer.

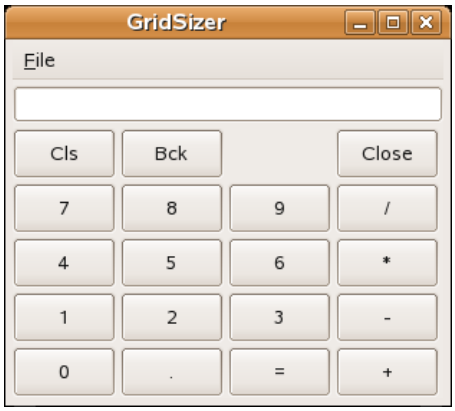


Figure: GridSizer example

```
#!/usr/bin/python

# gridsizer.py

import wx

class GridSizer(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 250))


        menubar = wx.MenuBar()
        file = wx.Menu()
        file.Append(1, '&Quit', 'Exit Calculator')
        menubar.Append(file, '&File')
        self.SetMenuBar(menubar)
```

```
        self.Bind(wx.EVT_MENU, self.OnClose, id=1)

        sizer = wx.BoxSizer(wx.VERTICAL)
        self.display = wx.TextCtrl(self, -1, '',  style=wx.TE_RIGHT)
        sizer.Add(self.display, 0, wx.EXPAND | wx.TOP | wx.BOTTOM, 4)
        gs = wx.GridSizer(4, 4, 3, 3)

        gs.AddMany( [(wx.Button(self, -1, 'Cls'), 0, wx.EXPAND),
            (wx.Button(self, -1, 'Bck'), 0, wx.EXPAND),
            (wx.StaticText(self, -1, ''), 0, wx.EXPAND),
            (wx.Button(self, -1, 'Close'), 0, wx.EXPAND),
            (wx.Button(self, -1, '7'), 0, wx.EXPAND),
            (wx.Button(self, -1, '8'), 0, wx.EXPAND),
            (wx.Button(self, -1, '9'), 0, wx.EXPAND),
            (wx.Button(self, -1, '/'), 0, wx.EXPAND),
            (wx.Button(self, -1, '4'), 0, wx.EXPAND),
            (wx.Button(self, -1, '5'), 0, wx.EXPAND),
            (wx.Button(self, -1, '6'), 0, wx.EXPAND),
            (wx.Button(self, -1, '*'), 0, wx.EXPAND),
            (wx.Button(self, -1, '1'), 0, wx.EXPAND),
            (wx.Button(self, -1, '2'), 0, wx.EXPAND),
            (wx.Button(self, -1, '3'), 0, wx.EXPAND),
            (wx.Button(self, -1, '-'), 0, wx.EXPAND),
            (wx.Button(self, -1, '0'), 0, wx.EXPAND),
            (wx.Button(self, -1, '.'), 0, wx.EXPAND),
            (wx.Button(self, -1, '='), 0, wx.EXPAND),
            (wx.Button(self, -1, '+'), 0, wx.EXPAND) ])

        sizer.Add(gs, 1, wx.EXPAND)
        self.SetSizer(sizer)
        self.Centre()
        self.Show(True)

    def OnClose(self, event):
        self.Close()

app = wx.App()
GridSizer(None, -1, 'GridSizer')
app.MainLoop()
```

Notice how we managed to put a space between the Bck and the Close buttons. We simply put an empty wx.StaticText there. Such tricks are quite common.

In our example we used the AddMany() method. It is a convenience method for adding multiple widgets at one time.

```
AddMany(list items)
```

Widgets are placed inside the table in the order, they are added. The first row is filled first, then the second row etc.

wx.FlexGridSizer

This sizer is similar to *wx.GridSizer*. It does also lay out it's widgets in a two dimensional table. It adds some flexibility to it. *wx.GridSizer* cells are of the same size. All cells in *wx.FlexGridSizer* have the same height in a row. All cells have the same width in a column. But all rows and columns are not necessarily the same height or width.

```
wx.FlexGridSizer(int rows=1, int cols=0, int vgap=0, int hgap=0)
```

rows and *cols* specify the number of rows and columns in a sizer. *vgap* and *hgap* add some space between widgets in both directions.

Many times developers have to develop dialogs for data input and modification. I find *wx.FlexGridSizer* suitable for such a task. A developer can easily set up a dialog

window with this sizer. It is also possible to accomplish this with *wx.GridSizer*, but it would not look nice, because of the constraint that each cell has the same size.

```
#!/usr/bin/python

# flexgridsizer.py

import wx

class FlexGridSizer(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(290, 250))

        panel = wx.Panel(self, -1)

        hbox = wx.BoxSizer(wx.HORIZONTAL)

        fgs = wx.FlexGridSizer(3, 2, 9, 25)

        title = wx.StaticText(panel, -1, 'Title')
        author = wx.StaticText(panel, -1, 'Author')
        review = wx.StaticText(panel, -1, 'Review')

        tc1 = wx.TextCtrl(panel, -1)
        tc2 = wx.TextCtrl(panel, -1)
        tc3 = wx.TextCtrl(panel, -1, style=wx.TE_MULTILINE)

        fgs.AddMany([(title), (tc1, 1, wx.EXPAND), (author), (tc2, 1, wx.EXPAND),
            (review, 1, wx.EXPAND), (tc3, 1, wx.EXPAND)])

        fgs.AddGrowableRow(2, 1)
        fgs.AddGrowableCol(1, 1)

        hbox.Add(fgs, 1, wx.ALL | wx.EXPAND, 15)
        panel.SetSizer(hbox)

        self.Centre()
        self.Show(True)

app = wx.App()
FlexGridSizer(None, -1, 'FlexGridSizer')
app.MainLoop()
```

```
hbox = wx.BoxSizer(wx.HORIZONTAL)
...
hbox.Add(fgs, 1, wx.ALL | wx.EXPAND, 15)
```

We create a horizontal box sizer in order to put some space (15px) around the table of widgets.

```
fgs.AddMany([(title), (tc1, 1, wx.EXPAND), (author), (tc2, 1, wx.EXPAND),
     (review, 1, wx.EXPAND), (tc3, 1, wx.EXPAND)])
```

We add widgets to the sizer with the *AddMany()* method. Both *wx.FlexGridSizer* and *wx.GridSizer* share this method.

```
fgs.AddGrowableRow(2, 1)
fgs.AddGrowableCol(1, 1)
```

We make the third row and second column growable. This way we let the text controls grow, when the window is resized. The first two text controls will grow in horizontal direction, the third one will grow in both direction. We must not forget to make the widgets expandable (wx.EXPAND) in order to make it really work.

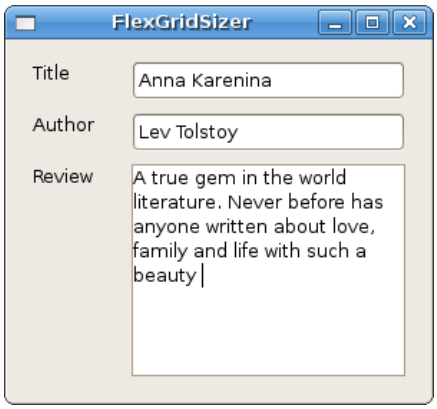


Figure: FlexGridSizer example

wx.GridBagSizer

The most complicated sizer in wxPython. Many programmer find it difficult to use. This kind of sizer is not typical only for wxPython. We can find it in other toolkits as well. There is no magic in using this sizer. Even though it is more complicated, it is certainly not rocket science. All we have to do is to create several layouts with it. Find all the quirks. Play with it a bit. There are more difficult things in programming. Believe me.

This sizer enables explicit positioning of items. Items can also optionally span more than one row and/or column. wx.GridBagSizer has a simple constructor.

```
wx.GridBagSizer(integer vgap, integer hgap)
```

The vertical and the horizontal gap defines the space in pixels used among all children. We add items to the grid with the Add() method.

```
Add(self, item, tuple pos, tuple span=wx.DefaultSpan, integer flag=0, integer border=0, userData=
```

Item is a widget that you insert into the grid. pos specifies the position in the virtual grid. The topleft cell has pos of (0, 0). span is an optional spanning of the widget. e.g. span of (3, 2) spans a widget across 3 rows and 2 columns. flag and border were discussed earlier by wx.BoxSizer. The items in the grid can change their size or keep the default size, when the window is resized. If you want your items to grow and shrink, you can use these two methods.

```
AddGrowableRow(integer row)
AddGrowableCol(integer col)
```

Rename dialog

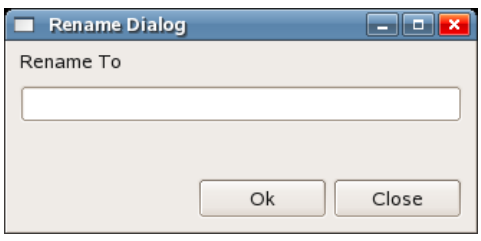


Figure: Rename window

The first example is intentionally a very simple one. So that it could be easily understood. There is no need to be afraid of wx.GridBagSizer. Once you understand it's logic, it is quite simple to use it. In our example, we will create a rename dialog. It will have one wx.StaticText, one wx.TextCtrl and two wx.Button-s.

```
#!/usr/bin/python

# rename.py

import wx

class Rename(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(320, 130))

        panel = wx.Panel(self, -1)
        sizer = wx.GridBagSizer(4, 4)

        text = wx.StaticText(panel, -1, 'Rename To')
        sizer.Add(text, (0, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)

        tc = wx.TextCtrl(panel, -1)
        sizer.Add(tc, (1, 0), (1, 5), wx.EXPAND | wx.LEFT | wx.RIGHT, 5)

        buttonOk = wx.Button(panel, -1, 'Ok', size=(90, 28))
        buttonClose = wx.Button(panel, -1, 'Close', size=(90, 28))
        sizer.Add(buttonOk, (3, 3))
        sizer.Add(buttonClose, (3, 4), flag=wx.RIGHT | wx.BOTTOM, border=5)

        sizer.AddGrowableCol(1)
        sizer.AddGrowableRow(2)
        panel.SetSizerAndFit(sizer)
        self.Centre()
        self.Show(True)


app = wx.App()
Rename(None, -1, 'Rename Dialog')
app.MainLoop()
```

We must look at the dialog window as a one big grid table.

```
text = wx.StaticText(panel, -1, 'Rename To')
sizer.Add(text, (0, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)
```

The text 'Rename to' goes to the left upper corner. So we specify the (0, 0) position. Plus we add some space to the bottom, left and bottom.

```
tc = wx.TextCtrl(panel, -1)
sizer.Add(tc, (1, 0), (1, 5), wx.EXPAND | wx.LEFT | wx.RIGHT, 5)
```

The wx.TextCtrl goes to the beginning of the second row (1, 0). Remember, that we count from zero. It expands 1 row and 5 columns. (1, 5). Plus we put 5 pixels of space to the left and to the right of the widget.

```
sizer.Add(buttonOk, (3, 3))
sizer.Add(buttonClose, (3, 4), flag=wx.RIGHT | wx.BOTTOM, border=5)
```

We put two buttons into the fourth row. The third row is left empty, so that we have some space between the wx.TextCtrl and the buttons. We put the ok button into the fourth column and the close button into the fifth one. Notice that once we apply some

space to one widget, it is applied to the whole row. That's why I did not specify bottom space for the ok button. A careful reader might notice, that we did not specify any space between the two buttons. e.g. we did not put any space to the right of the ok button, or to the right of the close button. In the constructor of the wx.GridBagSizer, we put some space between all widgets. So there is some space already.

```
sizer.AddGrowableCol(1)
sizer.AddGrowableRow(2)
```

The last thing we must do, is to make our dialog resizable. We make the second column and the third row growable. Now we can expand or shrink our window. Try to comment those two lines and see what happens.

Open Resource

The next example will be a bit more complicated. We will create an Open Resource window. This example will show a layout of a very handy dialog which you can find in Eclipse IDE.

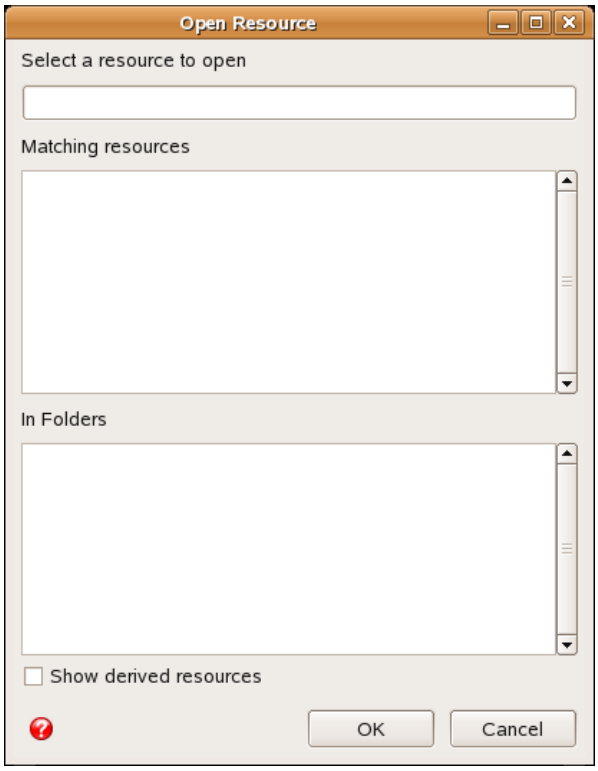


Figure: Open Resource window

```
#!/usr/bin/python

# openresource.py

import wx

class OpenResource(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(400, 500))

        panel = wx.Panel(self, -1)
        sizer = wx.GridBagSizer(4, 4)

        text1 = wx.StaticText(panel, -1, 'Select a resource to open')
```

```
        sizer.Add(text1, (0, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)

        tc = wx.TextCtrl(panel, -1)
        sizer.Add(tc, (1, 0), (1, 3), wx.EXPAND | wx.LEFT | wx.RIGHT, 5)

        text2 = wx.StaticText(panel, -1, 'Matching resources')
        sizer.Add(text2, (2, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)

        list1 = wx.ListBox(panel, -1, style=wx.LB_ALWAYS_SB)
        sizer.Add(list1, (3, 0), (5, 3), wx.EXPAND | wx.LEFT | wx.RIGHT, 5)

        text3 = wx.StaticText(panel, -1, 'In Folders')
        sizer.Add(text3, (8, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=5)

        list2 = wx.ListBox(panel, -1, style=wx.LB_ALWAYS_SB)
        sizer.Add(list2, (9, 0), (3, 3), wx.EXPAND | wx.LEFT | wx.RIGHT, 5)

        cb = wx.CheckBox(panel, -1, 'Show derived resources')
        sizer.Add(cb, (12, 0), flag=wx.LEFT | wx.RIGHT, border=5)

        buttonOk = wx.Button(panel, -1, 'OK', size=(90, 28))
        buttonCancel = wx.Button(panel, -1, 'Cancel', size=(90, 28))
        sizer.Add(buttonOk, (14, 1))
        sizer.Add(buttonCancel, (14, 2), flag=wx.RIGHT | wx.BOTTOM, border=5)

        help = wx.BitmapButton(panel, -1, wx.Bitmap('icons/help16.png'), style=wx.NO_BORDER)
        sizer.Add(help, (14, 0), flag=wx.LEFT, border=5)

        sizer.AddGrowableCol(0)
        sizer.AddGrowableRow(3)
        sizer.AddGrowableRow(9)
        sizer.SetEmptyCellSize((5, 5))
        panel.SetSizer(sizer)

        self.Centre()
        self.Show(True)

app = wx.App()
OpenResource(None, -1, 'Open Resource')
app.MainLoop()
```

```
sizer.AddGrowableRow(3)
sizer.AddGrowableRow(9)
```

We want to have both wx.ListBox-es growable. So we make the first row of each wx.ListBox growable.

Create new class

newclass.py example is a type of a window, that I found in JDeveloper. It is a dialog window for creating a new class in Java.

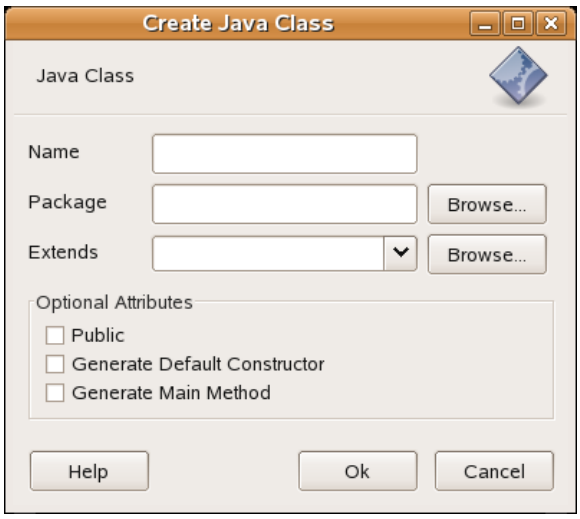


Figure: new class window

```
#!/usr/bin/python

# newclass.py

import wx

class NewClass(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        panel = wx.Panel(self, -1)
        sizer = wx.GridBagSizer(0, 0)

        text1 = wx.StaticText(panel, -1, 'Java Class')
        sizer.Add(text1, (0, 0), flag=wx.TOP | wx.LEFT | wx.BOTTOM, border=15)

        icon = wx.StaticBitmap(panel, -1, wx.Bitmap('icons/exec.png'))
        sizer.Add(icon, (0, 4), flag=wx.LEFT,  border=45)

        line = wx.StaticLine(panel, -1 )
        sizer.Add(line, (1, 0), (1, 5), wx.TOP | wx.EXPAND, -15)

        text2 = wx.StaticText(panel, -1, 'Name')
        sizer.Add(text2, (2, 0), flag=wx.LEFT, border=10)

        tc1 = wx.TextCtrl(panel, -1, size=(-1, 30))
        sizer.Add(tc1, (2, 1), (1, 3), wx.TOP | wx.EXPAND, -5)

        text3 = wx.StaticText(panel, -1, 'Package')
        sizer.Add(text3, (3, 0), flag= wx.LEFT | wx.TOP, border=10)

        tc2 = wx.TextCtrl(panel, -1)
        sizer.Add(tc2, (3, 1), (1, 3), wx.TOP | wx.EXPAND, 5)

        button1 = wx.Button(panel, -1, 'Browse...', size=(-1, 30))
        sizer.Add(button1, (3, 4), (1, 1), wx.TOP | wx.LEFT | wx.RIGHT , 5)

        text4 = wx.StaticText(panel, -1, 'Extends')
        sizer.Add(text4, (4, 0), flag=wx.TOP | wx.LEFT, border=10)

        combo = wx.ComboBox(panel, -1, )
        sizer.Add(combo, (4, 1), (1, 3), wx.TOP | wx.EXPAND,  5)

        button2 = wx.Button(panel, -1, 'Browse...', size=(-1, 30))
        sizer.Add(button2, (4, 4), (1, 1), wx.TOP | wx.LEFT | wx.RIGHT , 5)

        sb = wx.StaticBox(panel, -1, 'Optional Attributes')
        boxsizer = wx.StaticBoxSizer(sb, wx.VERTICAL)
        boxsizer.Add(wx.CheckBox(panel, -1, 'Public'), 0, wx.LEFT | wx.TOP, 5)
        boxsizer.Add(wx.CheckBox(panel, -1, 'Generate Default Constructor'), 0,  wx.LEFT, 5)
        boxsizer.Add(wx.CheckBox(panel, -1, 'Generate Main Method'), 0, wx.LEFT | wx.BOTTOM, 5)
        sizer.Add(boxsizer, (5, 0), (1, 5), wx.EXPAND | wx.TOP | wx.LEFT | wx.RIGHT , 10)
        button3 = wx.Button(panel, -1, 'Help', size=(-1, 30))
        sizer.Add(button3, (7, 0), (1, 1),  wx.LEFT, 10)

        button4 = wx.Button(panel, -1, 'Ok', size=(-1, 30))
        sizer.Add(button4, (7, 3), (1, 1),  wx.LEFT, 10)

        button5 = wx.Button(panel, -1, 'Cancel', size=(-1, 30))
        sizer.Add(button5, (7, 4), (1, 1),  wx.LEFT | wx.BOTTOM | wx.RIGHT, 10)

        sizer.AddGrowableCol(2)
        sizer.Fit(self)
        panel.SetSizer(sizer)
        self.Centre()
        self.Show(True)


app = wx.App()
```

```
NewClass(None, -1, 'Create Java Class')
app.MainLoop()
```

```
line = wx.StaticLine(panel, -1 )
sizer.Add(line, (1, 0), (1, 5), wx.TOP | wx.EXPAND, -15)
```

Notice, that we have used negative number for setting the top border. We could use wx.BOTTOM with border 15. We would get the same result.

```
icon = wx.StaticBitmap(panel, -1, wx.Bitmap('icons/exec.png'))
sizer.Add(icon, (0, 4), flag=wx.LEFT,  border=45)
```

We put an wx.StaticBitmap into the first row of the grid. We place it on the right side of the row. By using images we can make our applications look better.

```
sizer.Fit(self)
```

We did not set the size of the window explicitly. If we call Fit() method, the size of the window will exactly cover all widgets available. Try to comment this line and see what happens.

Home ǂ Contents ǂ Top of Page

Events in wxPython

Events are integral part of every GUI application. All GUI applications are event-driven. An application reacts to different event types which are generated during it's life. Events are generated mainly by the user of an application. But they can be generated by other means as well. e.g. internet connection, window manager, timer. So when we call MainLoop() method, our application waits for events to be generated. The MainLoop() method ends when we exit the application.

Definitions

Event is a piece of application-level information from the underlying framework, typically the GUI toolkit. **Event loop** is a programming construct that waits for and dispatches events or messages in a program. The event loop repeatedly looks for events to process them. A **dispatcher** is a process which maps events to **event handlers**. Event handlers are methods that react to events.

Event object is an object associated with the event. It is usually a window. **Event type** is a unique event, that has been generated. **Event binder** is an object, that binds an event type with an event handler.

A simple event example

In the following section we will describe a simple event. We will talk about a move event.

A move event is generated, when we move a window to a new position. The event type is **wx.MoveEvent**. The event binder for this event is **wx.EVT_MOVE**

```
#!/usr/bin/python

# moveevent.py
```

```
import wx

class MoveEvent(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 180))

        wx.StaticText(self, -1, 'x:', (10,10))
        wx.StaticText(self, -1, 'y:', (10,30))
        self.st1 = wx.StaticText(self, -1, '', (30, 10))
        self.st2 = wx.StaticText(self, -1, '', (30, 30))

        self.Bind(wx.EVT_MOVE, self.OnMove)

        self.Centre()
        self.Show(True)

    def OnMove(self, event):
        x, y = event.GetPosition()
        self.st1.SetLabel(str(x))
        self.st2.SetLabel(str(y))


app = wx.App()
MoveEvent(None, -1, 'move event')
app.MainLoop()
```

The example displays the current position of the window.

```
self.Bind(wx.EVT_MOVE, self.OnMove)
```

Here we bind the *wx.EVT_MOVE* event binder to the *OnMove()* method.

```
def OnMove(self, event):
    x, y = event.GetPosition()
```

The event parameter in the *OnMove()* method is an object specific to a particular event type. In our case it is the instance of a *wx.MoveEvent* class. This object holds information about the event. For example the Event object or the position of the window. In our case the Event object is the wx.Frame widget. We can find out the current position by calling the *GetPosition()* method of the event.

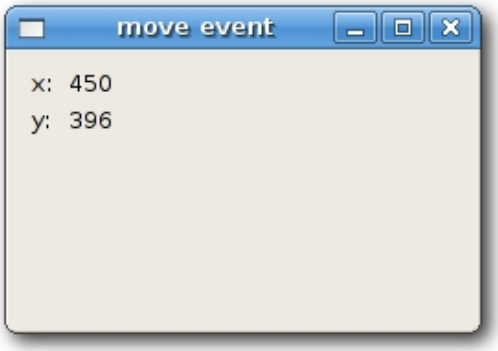


Figure: move event

Event binding

Working with events is straightforward in wxPython. There are three steps:

- Identify the event binder name: wx.EVT_SIZE, wx.EVT_CLOSE etc
- Create an event handler. It is a method, that is called, when an event is generated
- Bind an event to an event handler.

In wxPython we say to bind a method to an event. Sometimes a word hook is used. You bind an event by calling the Bind() method. The method has the following parameters:

```
Bind(event, handler, source=None, id=wx.ID_ANY, id2=wx.ID_ANY)
```

- event is one of EVT_* objects. It specifies the type of the event.
- handler is an object to be called. In other words, it is a method, that a programmer binds to an event.
- source parameter is used when we want to differentiate between the same event type from different widgets.
- id parameter is used, when we have multiple buttons, menu items etc. The id is used to differentiate among them.
- id2 is used when it is desirable to bind a handler to a range of ids, such as with EVT_MENU_RANGE.

Note that method Bind() is defined in class EvtHandler. It is the class, from which wx.Window inherits. wx.Window is a base class for most widgets in wxPython. There is also a reverse process. If

we want to unbind a method from an event, we call the Unbind() method. It has the same paremeters as the above one.

Vetoing events

Sometimes we need to stop processing an event. To do this, we call the method *Veto()*.

```
#!/usr/bin/python

# veto.py

import wx

class Veto(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 200))


        self.Bind(wx.EVT_CLOSE, self.OnClose)

        self.Centre()
        self.Show(True)

    def OnClose(self, event):

        dial = wx.MessageDialog(None, 'Are you sure to quit?', 'Question
            wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
        ret = dial.ShowModal()
        if ret == wx.ID_YES:
            self.Destroy()
        else:
            event.Veto()


app = wx.App()
Veto(None, -1, 'Veto')
app.MainLoop()
```

In our example, we process a *wx.CloseEvent*. This event is called, when we click the X button on the titlebar, press Alt + F4 or select close from the system menu. In many applications, we want to prevent from accidentally closing the window, if we made some changes. To do this, we must bind the *wx.EVT_CLOSE* event binder.

```
dial = wx.MessageDialog(None, 'Are you sure to quit?', 'Question',
    wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
ret = dial.ShowModal()
```

During the close event, we show a message dialog.

```
if ret == wx.ID_YES:
    self.Destroy()
else:
    event.Veto()
```

Depending on the return value, we destroy the window, or veto the event. Notice that to close the window, we must call the *Destroy()* method. By calling the *Close()* method, we would end up in an endless cycle.

Event propagation

There are two types of events. Basic events and command events. They differ in propagation. Event propagation is travelling of events from child widgets to parent widgets and grand parent widgets etc. Basic events do not propagate. Command events do propagate. For example *wx.CloseEvent* is a basic event. It does not make sense for this event to propagate to parent widgets.

By default, the event that is catched in a event handler stops propagating. To continue propagation, we must call the *Skip()* method.

```
#!/usr/bin/python

# propagate.py

import wx


class MyPanel(wx.Panel):
    def __init__(self, parent, id):
        wx.Panel.__init__(self, parent, id)

        self.Bind(wx.EVT_BUTTON, self.OnClicked)

    def OnClicked(self, event):
        print 'event reached panel class'
        event.Skip()


class MyButton(wx.Button):
    def __init__(self, parent, id, label, pos):
        wx.Button.__init__(self, parent, id, label, pos)
```

```
        self.Bind(wx.EVT_BUTTON, self.OnClicked)

    def OnClicked(self, event):
        print 'event reached button class'
        event.Skip()


class Propagate(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        panel = MyPanel(self, -1)

        MyButton(panel, -1, 'Ok', (15, 15))

        self.Bind(wx.EVT_BUTTON, self.OnClicked)

        self.Centre()
        self.Show(True)

    def OnClicked(self, event):
        print 'event reached frame class'
        event.Skip()


app = wx.App()
Propagate(None, -1, 'Propagate')
app.MainLoop()
```

In our example, we have a button on a panel. The panel is placed in a frame widget. We define a handler for all widgets.

```
 event reached button class
 event reached panel class
 event reached frame class
```

We get this, when we click on the button. The event travels from the button to panel and to frame.

Try to omit some Skip() methods and see, what hapens.

Window identifiers

Window identifiers are integers that uniquely determine the window identity in the event system. There are three ways to create window id's.

- let the system automatically create an id
- use standard identifiers
- create your own id

Each widget has an id parameter. This is a unique number in the event system. If we work with multiple widgets, we must differantiate among them.

```
wx.Button(parent, -1)
wx.Button(parent, wx.ID_ANY)
```

If we provide -1 or wx.ID_ANY for the id parameter, we let the wxPython automatically create an id for us. The automatically created id's are always negative, whereas user specified id's must always be positive. We usually use this option when we do not need to change the widget state. For example a static text, that will never be changed during the life of the application. We can still get the id, if we want. There is a method *GetId()*, which will determine the id for us.

```
#!/usr/bin/python

# automaticids.py

import wx

class AuIds(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(170, 100))

        panel = wx.Panel(self, -1)
        exit = wx.Button(panel, -1, 'Exit', (10, 10))

        self.Bind(wx.EVT_BUTTON,  self.OnExit, id=exit.GetId())

        self.Centre()
        self.Show(True)

    def OnExit(self, event):
        self.Close()


app = wx.App()
AuIds(None, -1, '')
app.MainLoop()
```

In this example, we do not care about the actual id value.

```
self.Bind(wx.EVT_BUTTON,  self.OnExit, id=exit.GetId())
```

We get the automatically generated id by calling the *GetId()* method.

Standard identifiers should be used whenever possible. The identifiers can provide some standard graphics or behaviour on some platforms.

```
#!/usr/bin/python

# identifiers.py

import wx

class Identifiers(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(200, 150))

        panel = wx.Panel(self, -1)
        grid = wx.GridSizer(3, 2)

        grid.AddMany([(wx.Button(panel, wx.ID_CANCEL), 0, wx.TOP | wx.LEI
            (wx.Button(panel, wx.ID_DELETE), 0, wx.TOP, 9),
            (wx.Button(panel, wx.ID_SAVE), 0, wx.LEFT, 9),
            (wx.Button(panel, wx.ID_EXIT)),
            (wx.Button(panel, wx.ID_STOP), 0, wx.LEFT, 9),
            (wx.Button(panel, wx.ID_NEW))])


        self.Bind(wx.EVT_BUTTON, self.OnQuit, id=wx.ID_EXIT)

        panel.SetSizer(grid)
        self.Centre()
        self.Show(True)

    def OnQuit(self, event):
        self.Close()

app = wx.App()
Identifiers(None, -1, '')
app.MainLoop()
```

In our example we use standard identifiers on buttons. On linux, the buttons have small icons.

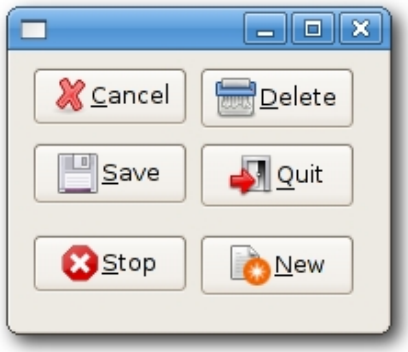


Figure: standard identifiers

The last option is to use own identifiers. We define our own global ids.

Miscellaneous events

Focus event

The focus indicates the currently selected widget in application. The text entered from the keyboard or pasted from the clipboard is sent to the widget, which has the focus. There are two event types concering focus. The **wx.EVT_SET_FOCUS** event, which is generated when a widget receives focus. The **wx.EVT_KILL_FOCUS** is generated, when the widget looses focus. The focus is changed by clicking or by a keybord key. Usually Tab/Shift+Tab.

```
#!/usr/bin/python

# focusevent.py

import wx


class MyWindow(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent, -1)

        self.color = '#b3b3b3'

        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_SET_FOCUS, self.OnSetFocus)
        self.Bind(wx.EVT_KILL_FOCUS, self.OnKillFocus)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.SetPen(wx.Pen(self.color))
        x, y = self.GetSize()
        dc.DrawRectangle(0, 0, x, y)

    def OnSize(self, event):
        self.Refresh()

    def OnSetFocus(self, event):
        self.color = '#0099f7'
        self.Refresh()
```

```
    def OnKillFocus(self, event):
        self.color = '#b3b3b3'
        self.Refresh()

class FocusEvent(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 250))

        grid = wx.GridSizer(2, 2, 10, 10)
        grid.AddMany([(MyWindow(self), 1, wx.EXPAND|wx.TOP|wx.LEFT,9),
            (MyWindow(self), 1, wx.EXPAND|wx.TOP|wx.RIGHT, 9),
            (MyWindow(self), 1, wx.EXPAND|wx.BOTTOM|wx.LEFT, 9),
            (MyWindow(self), 1, wx.EXPAND|wx.BOTTOM|wx.RIGHT, 9)])


        self.SetSizer(grid)
        self.Centre()
        self.Show(True)

app = wx.App()
FocusEvent(None, -1, 'focus event')
app.MainLoop()
```

In our example, we have four panels. The panel with focus is highlighted.

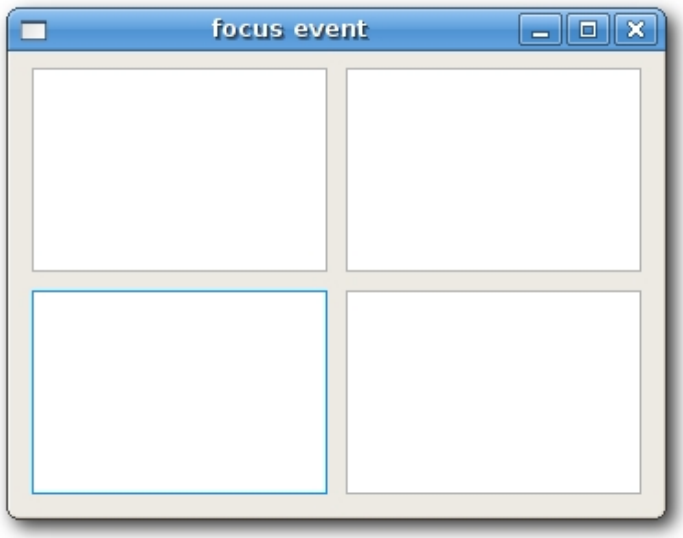


Figure: focus event

ScrollEvent

The following code is an example of a wx.ScrollWinEvent. This event is generated, when we click on a built in Scrollbar. Built-in Scrollbar is activated with the SetScrollbar() method call. For stand-alone Scrollbars, there is another event type, namely

wx.ScrollEvent.

```
#!/usr/bin/python

# myscrollwinevent.py

import wx

class ScrollWinEvent(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)
        panel = wx.Panel(self, -1)
        self.st = wx.StaticText(panel, -1, '0', (30,0))
        panel.Bind(wx.EVT_SCROLLWIN, self.OnScroll)
        panel.SetScrollbar(wx.VERTICAL, 0, 6, 50);
        self.Centre()
        self.Show(True)

    def OnScroll(self, evt):
        y = evt.GetPosition()
        self.st.SetLabel(str(y))

app = wx.App()
ScrollWinEvent(None, -1, 'scrollwinevent.py')
app.MainLoop()
```

SizeEvent

A wx.SizeEvent is generated, when our window is resized. In our example, we show the size of the window in the titlebar.

```
#!/usr/bin/python

# sizeevent.py

import wx

class SizeEvent(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Centre()
        self.Show(True)

    def OnSize(self, event):
        self.SetTitle(str(event.GetSize()))


app = wx.App()
SizeEvent(None, 1, 'sizeevent.py')
app.MainLoop()
```

```
self.SetTitle(str(event.GetSize()))
```

To get the current size of the window, we call the *GetSize()* method of the event object.

Figure: sizeevent.py

PaintEvent

A paint event is generated when a window is redrawn. This happens when we resize a window or when we maximize it. A paint event can be generated programatically as well. For example, when we call SetLabel() method to change a wx.StaticText widget. Note that when we minimize a window, no paint event is generated.

```
#!/usr/bin/python

# paintevent.py

import wx

class PaintEvent(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        self.count = 0
        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        self.count = self.count + 1
        print self.count
```

```
app = wx.App()
PaintEvent(None, -1, 'paintevent.py')
app.MainLoop()
```

In our example we print the number of paint events generated into the console.

KeyEvent

When we press a key on our keyboard, wx.KeyEvent is generated. This event is sent to the widget that has currently focus. There are three different key handlers:

- wx.EVT_KEY_DOWN
- wx.EVT_KEY_UP
- wx.EVT_CHAR

A common request is to close application, when Esc key is pressed.

```
#!/usr/bin/python

# keyevent.py

import wx

class KeyEvent(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        panel = wx.Panel(self, -1)
        panel.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)
        panel.SetFocus()

        self.Centre()
        self.Show(True)


    def OnKeyDown(self, event):
        keycode = event.GetKeyCode()
        if keycode == wx.WXK_ESCAPE:
            ret  = wx.MessageBox('Are you sure to quit?', 'Question',
                wx.YES_NO | wx.NO_DEFAULT, self)
            if ret == wx.YES:
                self.Close()
        event.Skip()


app = wx.App()
KeyEvent(None, -1, 'keyevent.py')
app.MainLoop()
```

```
keycode = event.GetKeyCode()
```

Here we get the key code of the pressed key.

```
if keycode == wx.WXK_ESCAPE:
```

We check the key code. The Esc key has *wx.WXK_ESCAPE* code.

Home ‡ Contents ‡ Top of Page

wxPython 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. 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.

A Simple message box

A message box provides short information to the user. A good example is a cd burning application. When a cd is finished burning, a message box pops up.

```
#!/usr/bin/python

# message.py

import wx

class MessageDialog(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        wx.FutureCall(5000, self.ShowMessage)

        self.Centre()
        self.Show(True)

    def ShowMessage(self):
        wx.MessageBox('Download completed', 'Info')


app = wx.App()
MessageDialog(None, -1, 'MessageDialog')
app.MainLoop()
```

```
wx.FutureCall(5000, self.ShowMessage)
```

wx.FutureCall calls a method after 5 seconds. The first parameter is a time value, after which a given method is called. The parameter is in milliseconds. The second parameter is a method to be called.

```
def ShowMessage(self):
    wx.MessageBox('Download completed', 'Info')
```

wx.MessageBox shows a small dialog window. We provide three parameters. The text message, the title message and finally the button.

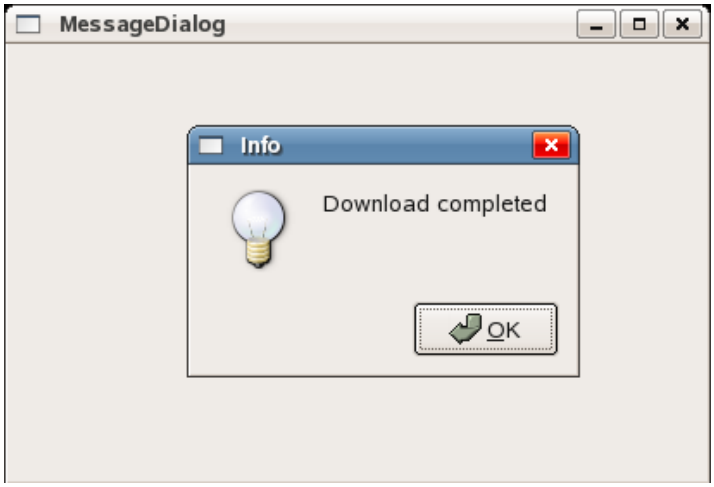


Figure: A message dialog

Predefined dialogs

wxPython has several predefined dialogs. These are dialogs for common programming tasks like showing text, receiving input , loading and saving files etc.

Message dialogs

Message dialogs are used to show messages to the user. They are more flexible than simple message boxes, that we saw in the previous example. They are customizable. We can change icons and buttons that will be shown in a dialog.

```
wx.MessageDialog(wx.Window parent, string message, string caption=wx.MessageB
 long style=wx.OK | wx.CANCEL | wx.CENTRE, wx.Point pos=(-1, -1))
```

flag	meaning
wx.OK	show Ok button
wx.CANCEL	show Cancel button
wx.YES_NO	show Yes, No buttons

wx.YES_DEFAULT	make Yes button the default
wx.NO_DEFAULT	make No button the default
wx.ICON_EXCLAMATION	show an alert icon
wx.ICON_ERROR	show an error icon
wx.ICON_HAND	same as wx.ICON_ERROR
wx.ICON_INFORMATION	show an info icon
wx.ICON_QUESTION	show a question icon

```
#!/usr/bin/python

# messages.py

import wx

class Messages(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        panel = wx.Panel(self, -1)

        hbox = wx.BoxSizer()
        sizer = wx.GridSizer(2, 2, 2, 2)

        btn1 = wx.Button(panel, -1, 'Info')
        btn2 = wx.Button(panel, -1, 'Error')
        btn3 = wx.Button(panel, -1, 'Question')
        btn4 = wx.Button(panel, -1, 'Alert')

        sizer.AddMany([btn1, btn2, btn3, btn4])

        hbox.Add(sizer, 0, wx.ALL, 15)
        panel.SetSizer(hbox)


        btn1.Bind(wx.EVT_BUTTON, self.ShowMessage1)
        btn2.Bind(wx.EVT_BUTTON, self.ShowMessage2)
        btn3.Bind(wx.EVT_BUTTON, self.ShowMessage3)
        btn4.Bind(wx.EVT_BUTTON, self.ShowMessage4)

        self.Centre()
        self.Show(True)

    def ShowMessage1(self, event):
        dial = wx.MessageDialog(None, 'Download completed', 'Info', wx.OK)
        dial.ShowModal()

    def ShowMessage2(self, event):
        dial = wx.MessageDialog(None, 'Error loading file', 'Error', wx.OK |
            wx.ICON_ERROR)
        dial.ShowModal()

    def ShowMessage3(self, event):
        dial = wx.MessageDialog(None, 'Are you sure to quit?', 'Question',
            wx.YES_NO | wx.NO_DEFAULT | wx.ICON_QUESTION)
        dial.ShowModal()
```

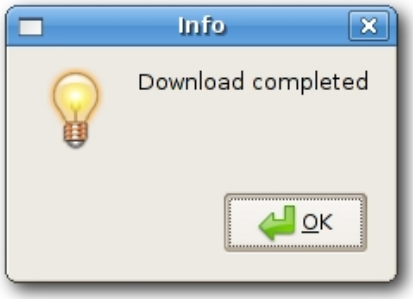
```
    def ShowMessage4(self, event):
        dial = wx.MessageDialog(None, 'Unallowed operation', 'Exclamation', \
            wx.ICON_EXCLAMATION)
        dial.ShowModal()

app = wx.App()
Messages(None, -1, 'Messages')
app.MainLoop()
```

In our example, we have created four buttons and put them in a grid sizer. These buttons will show four different dialog windows. We create them by specifying different style flags.

```
dial = wx.MessageDialog(None, 'Error loading file', 'Error', wx.OK |
    wx.ICON_ERROR)
dial.ShowModal()
```

The creation of the message dialog is simple. We set the dialog to be a toplevel window by providing None as a parent. The two strings provide the message text and the dialog title. We show an ok button and an error icon by specifying the *wx.OK* and *wx.ICON_ERROR* flags. To show the dialog on screen, we call the *ShowModal()* method.



About dialog box

Almost every application has a typical about dialog box. It is usually placed in the Help menu. The purpose of this dialog is to give the user the basic information about the name and the version of the application. In the past, these dialogs used to be quite brief. These days most of these boxes provide additional information about the authors. They give credits to additional programmers or documentation writers. They also provide information about the application licence. These boxes can show the logo of the company or the application logo. Some of the more capable about boxes show animation. wxPython has a special about dialog box starting from 2.8.x series. It is not rocket science to make such a dialog manually. But it makes a programmer's life easier.

The dialog box is located in the Misc module. In order to create an about dialog box we must create two objects. A *wx.AboutDialogInfo* and a *wx.AboutBox*.

```
wx.AboutDialogInfo()
```

We will call the following methods upon a *wx.AboutDialogInfo* object in our example. These methods are self-exlanatory.

Method	Description
SetName(string name)	set the name of the program
SetVersion(string version)	set the version of the program
SetDescription(string desc)	set the description of the program
SetCopyright(string copyright)	set the copyright fo the program
SetLicence(string licence)	set the licence of the program
SetIcon(wx.Icon icon)	set the icon to be show
SetWebSite(string URL)	set the website of the program
SetLicence(string licence)	set the licence of the program

AddDeveloper(string developer)	add a developer to the developer's list
AddDocWriter(string docwirter)	add a docwriter to the docwriter's list
AddArtist(string artist)	add an artist to the artist's list
AddTranslator(string developer)	add a developer to the translator's list

The constructor of the *wx.AboutBox* is as follows. It takes a *wx.AboutDialogInfo* as a parameter.

```
wx.AboutBox(wx.AboutDialogInfo info)
```

wxPython can display two kinds of About boxes. It depends on which platform we use and which methods we call. It can be a native dialog or a wxPython generic dialog. Windows native about dialog box cannot display custom icons, licence text nor the url's. If we omit these three fields, wx.Python will show a native dialog. Otherwise it will resort to a generic one. It is advised to provide licence information in a separate menu item, if we want to stay as native as possible. GTK+ can show all these fields.

```
#!/usr/bin/python

# aboutbox.py

import wx

ID_ABOUT = 1

class AboutDialogBox(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(260, 200))

        menubar = wx.MenuBar()
        help = wx.Menu()
        help.Append(ID_ABOUT, '&About')
        self.Bind(wx.EVT_MENU, self.OnAboutBox, id=ID_ABOUT)
        menubar.Append(help, '&Help')
        self.SetMenuBar(menubar)

        self.Centre()
        self.Show(True)

    def OnAboutBox(self, event):
        description = """File Hunter is an advanced file manager for the Uni
system. Features include powerful built-in editor, advanced search capabilit
powerful batch renaming, file comparison, extensive archive handling and mor
"""

        licence = """File Hunter is free software; you can redistribute it a
```

```
under the terms of the GNU General Public License as published by the Free S
either version 2 of the License, or (at your option) any later version.

File Hunter is distributed in the hope that it will be useful, but WITHOUT A
without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTIC
See the GNU General Public License for more details. You should have receive
the GNU General Public License along with File Hunter; if not, write to
the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  

        info = wx.AboutDialogInfo()

        info.SetIcon(wx.Icon('icons/hunter.png', wx.BITMAP_TYPE_PNG))
        info.SetName('File Hunter')
        info.SetVersion('1.0')
        info.SetDescription(description)
        info.SetCopyright('(C) 2007 jan bodnar')
        info.SetWebSite('http://www.zetcode.com')
        info.SetLicence(licence)
        info.AddDeveloper('jan bodnar')
        info.AddDocWriter('jan bodnar')
        info.AddArtist('The Tango crew')
        info.AddTranslator('jan bodnar')

        wx.AboutBox(info)


app = wx.App()
AboutDialogBox(None, -1, 'About dialog box')
app.MainLoop()
```

```
        description = """File Hunter is an advanced file manager for the Unix
system. Features include powerful built-in editor, advanced search capabilitie
powerful batch renaming, file comparison, extensive archive handling and more.
"""
```

It is not the best idea to put too much text into the code of the application. I don't want to make the example too complex, so I put all the text into the code. But in real world programs, the text should be placed separately inside a file. It helps us with the maintenace of our application. For example, if we want to translate our application to other languages.

```
info = wx.AboutDialogInfo()
```

The first thing to do is to create a *wx.AboutDialogInfo* object. The constructor is empty. It does not taky any parameters.

```
info.SetIcon(wx.Icon('icons/hunter.png', wx.BITMAP_TYPE_PNG))
info.SetName('File Hunter')
info.SetVersion('1.0')
info.SetDescription(description)
info.SetCopyright('(C) 2007 jan bodnar')
```

```
info.SetWebSite('http://www.zetcode.com')
info.SetLicence(licence)
info.AddDeveloper('jan bodnar')
info.AddDocWriter('jan bodnar')
info.AddArtist('The Tango crew')
info.AddTranslator('jan bodnar')
```

The next thing to do is to call all necessary methods upon the created *wx.AboutDialogInfo* object.

```
wx.AboutBox(info)
```

In the end we create a *wx.AboutBox* widget. The only parameter it takes is the *wx.AboutDialogInfo* object.

And of course, if we want to have an animation or some other eye candy, we must implement our about dialog manually.

About dialog box

A custom dialog

In the next example we create a custom dialog. An image editing application can change a color depth of a picture. To provide this funcionality, we could create a suitable dialog.

```
#!/usr/bin/python

# colordepth.py

import wx

ID_DEPTH = 1

class ChangeDepth(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(250, 210))

        panel = wx.Panel(self, -1)
        vbox = wx.BoxSizer(wx.VERTICAL)

        wx.StaticBox(panel, -1, 'Colors', (5, 5), (240, 150))
        wx.RadioButton(panel, -1, '256 Colors', (15, 30), style=wx.RB_GROUP)
        wx.RadioButton(panel, -1, '16 Colors', (15, 55))
        wx.RadioButton(panel, -1, '2 Colors', (15, 80))
        wx.RadioButton(panel, -1, 'Custom', (15, 105))
        wx.TextCtrl(panel, -1, '', (95, 105))

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        okButton = wx.Button(self, -1, 'Ok', size=(70, 30))
        closeButton = wx.Button(self, -1, 'Close', size=(70, 30))
        hbox.Add(okButton, 1)
        hbox.Add(closeButton, 1, wx.LEFT, 5)

        vbox.Add(panel)
        vbox.Add(hbox, 1, wx.ALIGN_CENTER | wx.TOP | wx.BOTTOM, 10)

        self.SetSizer(vbox)


class ColorDepth(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 220))

        toolbar = self.CreateToolBar()
        toolbar.AddLabelTool(ID_DEPTH, '', wx.Bitmap('icons/color.png'))

        self.Bind(wx.EVT_TOOL, self.OnChangeDepth, id=ID_DEPTH)

        self.Centre()
        self.Show(True)

    def OnChangeDepth(self, event):
        chgdep = ChangeDepth(None, -1, 'Change Color Depth')
        chgdep.ShowModal()
        chgdep.Destroy()

app = wx.App()
ColorDepth(None, -1, '')
app.MainLoop()
```

```
class ChangeDepth(wx.Dialog):
```

```
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(250, 210))
```

In our code example we create a custom ChangeDepth dialog. We inherit from a *wx.Dialog* widget.

```
chgdep = ChangeDepth(None, -1, 'Change Color Depth')
chgdep.ShowModal()
chgdep.Destroy()
```

We instantiate a ChangeDepth class. Then we call the *ShowModal()* dialog. We must not forget to destroy our dialog. Notice the visual difference between the dialog and the top level window. The dialog in the following figure has been activated. We cannot work with the toplevel window until the dialog is destroyed. There is a clear difference in the titlebar of the windows.

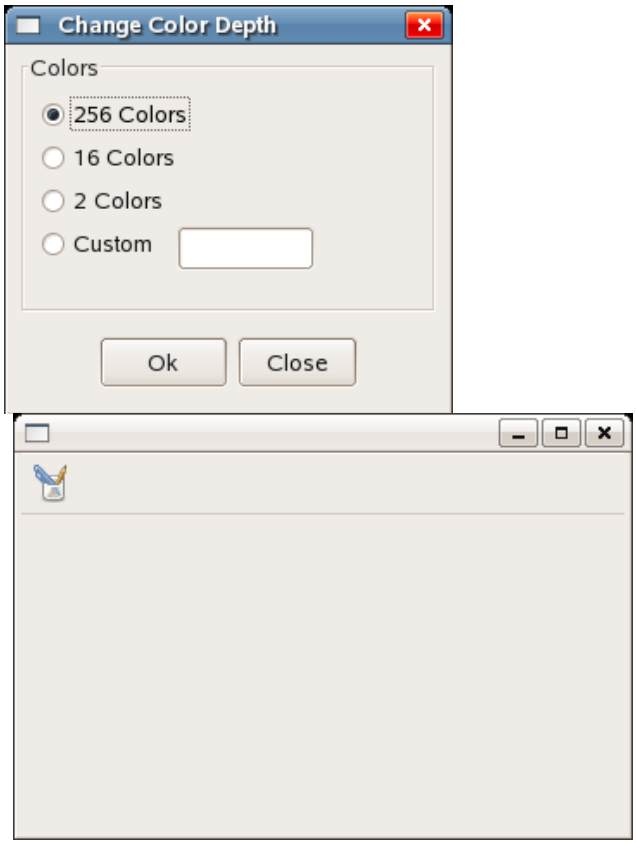


Figure: A custom dialog

Home ǂ Contents ǂ Top of Page

last modified June 5, 2007

Core Widgets

In this section, we will introduce basic widgets in wxPython. Each widget will have a small code example.

- wx.Button
- wx.ToggleButton
- wx.BitmapButton
- wx.StaticLine
- wx.StaticText
- wx.StaticBox
- wx.ComboBox
- wx.CheckBox
- wx.StatusBar
- wx.RadioButton
- wx.Gauge
- wx.Slider
- wx.ListBox
- wx.SpinCtrl
- wx.SplitterWindow
- wx.ScrolledWindow
- wx.Notebook
- wx.Panel

wx.Button

wx.Button is a simple widget. It contains a text string. It is used to trigger an action.

wx.Button styles

- wx.BU_LEFT
- wx.BU_TOP
- wx.BU_RIGHT
- wx.BU_BOTTOM
- wx.BU_EXACTFIT
- wx.NO_BORDER

Figure: Buttons.py

```
#!/usr/bin/python

# buttons.py

import wx
import random

APP_SIZE_X = 300
APP_SIZE_Y = 200

class MyButtons(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(APP_SIZE_X, APP

        wx.Button(self, 1, 'Close', (50, 130))
        wx.Button(self, 2, 'Random Move', (150, 130), (110, -1))

        self.Bind(wx.EVT_BUTTON, self.OnClose, id=1)
        self.Bind(wx.EVT_BUTTON, self.OnRandomMove, id=2)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnClose(self, event):
        self.Close(True)

    def OnRandomMove(self, event):
        screensize = wx.GetDisplaySize()
        randx = random.randrange(0, screensize.x - APP_SIZE_X)
        randy = random.randrange(0, screensize.y - APP_SIZE_Y)
        self.Move((randx, randy))

app = wx.App(0)
MyButtons(None, -1, 'buttons.py')
app.MainLoop()
```

wx.ToggleButton

wx.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.

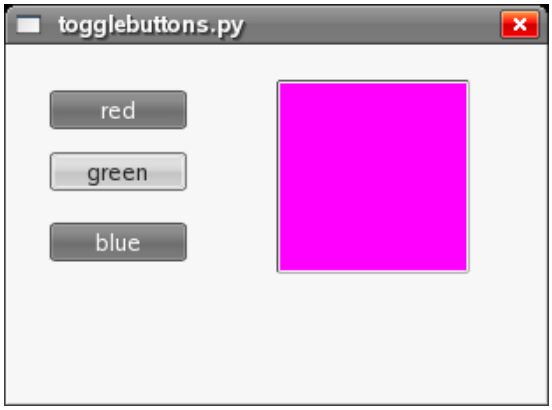


Figure: Togglebuttons.py

```
#!/usr/bin/python

# togglebuttons.py

import wx

class ToggleButtons(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(300, 200))

        self.colour = wx.Colour(0, 0, 0)

        wx.ToggleButton(self, 1, 'red', (20, 25))
        wx.ToggleButton(self, 2, 'green', (20, 60))
        wx.ToggleButton(self, 3, 'blue', (20, 100))

        self.panel  = wx.Panel(self, -1, (150, 20), (110, 110), style=wx.
        self.panel.SetBackgroundColour(self.colour)

        self.Bind(wx.EVT_TOGGLEBUTTON, self.ToggleRed, id=1)
        self.Bind(wx.EVT_TOGGLEBUTTON, self.ToggleGreen, id=2)
        self.Bind(wx.EVT_TOGGLEBUTTON, self.ToggleBlue, id=3)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def ToggleRed(self, event):
        green = self.colour.Green()
        blue = self.colour.Blue()
        if  self.colour.Red():
            self.colour.Set(0, green, blue)
        else:
            self.colour.Set(255, green, blue)
        self.panel.SetBackgroundColour(self.colour)
```

```
    def ToggleGreen(self, event):
        red = self.colour.Red()
        blue = self.colour.Blue()
        if  self.colour.Green():
            self.colour.Set(red, 0, blue)
        else:
            self.colour.Set(red, 255, blue)
        self.panel.SetBackgroundColour(self.colour)

    def ToggleBlue(self, event):
        red = self.colour.Red()
        green = self.colour.Green()
        if  self.colour.Blue():
            self.colour.Set(red, green, 0)
        else:
            self.colour.Set(red, green, 255)
        self.panel.SetBackgroundColour(self.colour)


app = wx.App()
ToggleButtons(None, -1, 'togglebuttons.py')
app.MainLoop()
```

wx.BitmapButton

A bitmap button is a button, that displays a bitmap. A bitmap button can have three other states. Selected, focused and displayed. We can set a specific bitmap for those states. A video player is a nice example, where bitmap buttons are used. We can see play, pause, next, previous and volume bitmap buttons there. So we create a skeleton of a video player in our next example.

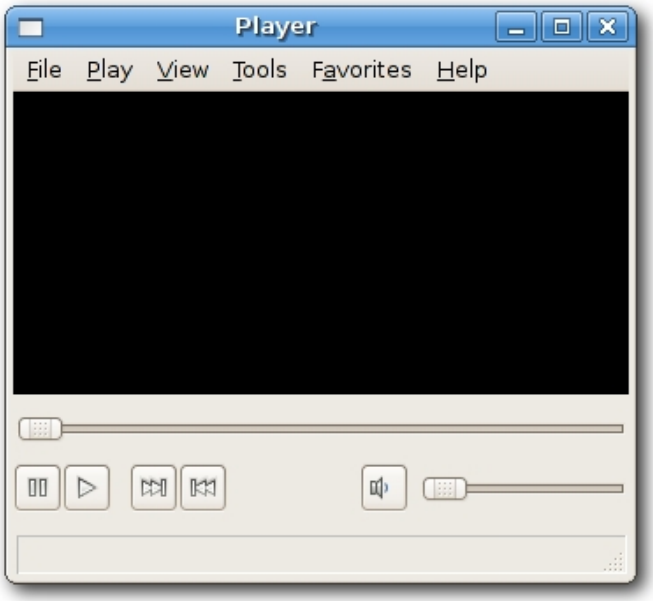


Figure: Player.py

```
#!/usr/bin/python

# player.py

import wx

class Player(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 300))
        panel = wx.Panel(self, -1)

        pnl1 = wx.Panel(self, -1)
        pnl1.SetBackgroundColour(wx.BLACK)
        pnl2 = wx.Panel(self, -1 )

        menubar = wx.MenuBar()
        file = wx.Menu()
        play = wx.Menu()
        view = wx.Menu()
        tools = wx.Menu()
        favorites = wx.Menu()
        help = wx.Menu()

        file.Append(101, '&quit', 'Quit application')

        menubar.Append(file, '&File')
        menubar.Append(play, '&Play')
        menubar.Append(view, '&View')
        menubar.Append(tools, '&Tools')
        menubar.Append(favorites, 'F&avorites')
        menubar.Append(help, '&Help')

        self.SetMenuBar(menubar)

        slider1 = wx.Slider(pnl2, -1, 0, 0, 1000)
        pause = wx.BitmapButton(pnl2, -1, wx.Bitmap('icons/stock_media-pa
        play  = wx.BitmapButton(pnl2, -1, wx.Bitmap('icons/stock_media-pl
        next  = wx.BitmapButton(pnl2, -1, wx.Bitmap('icons/stock_media-ne
        prev  = wx.BitmapButton(pnl2, -1, wx.Bitmap('icons/stock_media-pr
        volume = wx.BitmapButton(pnl2, -1, wx.Bitmap('icons/volume.png'))
        slider2 = wx.Slider(pnl2, -1, 0, 0, 100, size=(120, -1))

        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox1 = wx.BoxSizer(wx.HORIZONTAL)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)

        hbox1.Add(slider1, 1)
        hbox2.Add(pause)
        hbox2.Add(play, flag=wx.RIGHT, border=5)
        hbox2.Add(next, flag=wx.LEFT, border=5)
        hbox2.Add(prev)
        hbox2.Add((-1, -1), 1)
        hbox2.Add(volume)
        hbox2.Add(slider2, flag=wx.TOP | wx.LEFT, border=5)

        vbox.Add(hbox1, flag=wx.EXPAND | wx.BOTTOM, border=10)
        vbox.Add(hbox2, 1, wx.EXPAND)
        pnl2.SetSizer(vbox)
```

```
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(pnl1, 1, flag=wx.EXPAND)
        sizer.Add(pnl2, flag=wx.EXPAND | wx.BOTTOM | wx.TOP, border=10)

        self.SetMinSize((350, 300))
        self.CreateStatusBar()
        self.SetSizer(sizer)

        self.Centre()
        self.Show()


app = wx.App()
Player(None, -1, 'Player')
app.MainLoop()
```

```
pause = wx.BitmapButton(pnl2, -1, wx.Bitmap('icons/stock_media-pause.png'
```

The creation of the *wx.BitmapButton* is self explanatory.

```
hbox2.Add(prev)
hbox2.Add((-1, -1), 1)
hbox2.Add(volume)
```

Here we put some space between the previous button and the volume button. The proportion is set to 1. This means, that the space will grow, when we resize the window.

```
self.SetMinSize((350, 300))
```

Here we set the minimum size of the player. It does not make much sense to shrink the window below some value.

wx.StaticLine

This widget displays a simple line on the window. It can be horizontal or vertical. centraleurope.py script displays central european countries and their population. The wx.StatLine makes it look more visually attractive. wx.StaticLine styles

- wx.LI_HORIZONTAL
- wx.LI_VERTICAL

Figure: CentralEurope.py

```
#!/usr/bin/python

# centraleurope.py

import wx

class CentralEurope(wx.Dialog):
    def __init__ (self, parent, ID, title):
        wx.Dialog.__init__(self, parent, ID, title, size=(360, 370))

        font = wx.Font(10, wx.DEFAULT, wx.NORMAL, wx.BOLD)
        heading = wx.StaticText(self, -1, 'The Central Europe', (130, 15)
        heading.SetFont(font)

        wx.StaticLine(self, -1, (25, 50), (300,1))

        wx.StaticText(self, -1, 'Slovakia', (25, 80))
        wx.StaticText(self, -1, 'Hungary', (25, 100))
        wx.StaticText(self, -1, 'Poland', (25, 120))
        wx.StaticText(self, -1, 'Czech Republic', (25, 140))
        wx.StaticText(self, -1, 'Germany', (25, 160))
        wx.StaticText(self, -1, 'Slovenia', (25, 180))
        wx.StaticText(self, -1, 'Austria', (25, 200))
        wx.StaticText(self, -1, 'Switzerland', (25, 220))

        wx.StaticText(self, -1, '5 379 000', (250, 80))
        wx.StaticText(self, -1, '10 084 000', (250, 100))
        wx.StaticText(self, -1, '38 635 000', (250, 120))
        wx.StaticText(self, -1, '10 240 000', (250, 140))
        wx.StaticText(self, -1, '82 443 000', (250, 160))
        wx.StaticText(self, -1, '2 001 000', (250, 180))
```

```
        wx.StaticText(self, -1, '8 032 000', (250, 200))
        wx.StaticText(self, -1, '7 288 000', (250, 220))

        wx.StaticLine(self, -1, (25, 260), (300,1))

        sum = wx.StaticText(self, -1, '164 102 000', (240, 280))
        sum_font = sum.GetFont()
        sum_font.SetWeight(wx.BOLD)
        sum.SetFont(sum_font)

        wx.Button(self, 1, 'Ok', (140, 310), (60, 30))

        self.Bind(wx.EVT_BUTTON, self.OnOk, id=1)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnOk(self, event):
        self.Close()

app = wx.App()
CentralEurope(None, -1, 'centraleurope.py')
app.MainLoop()
```

wx.StaticText

A wx.StaticText widget displays one or more lines of read-only text. wx.StaticText Styles

- wx.ALIGN_RIGHT
  iwx.ALIGN_LEFT
- wx.ALIGN_CENTER / wx.ALIGN_CENTRE
- wx.ST_NO_AUTORESIZE

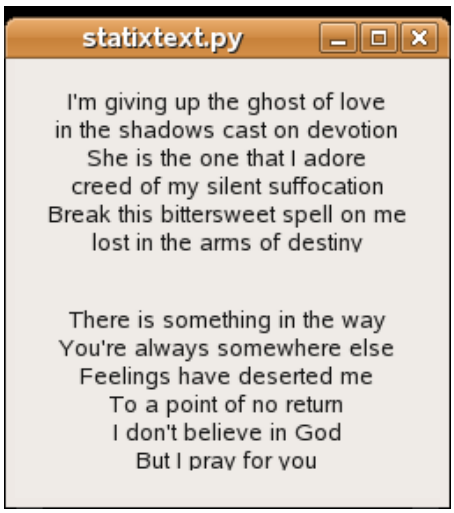


Figure: statictext.py

```
#!/usr/bin/python

# statictext.py

import wx

class StaticText(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        lyrics1 = '''I'm giving up the ghost of love
in the shadows cast on devotion
She is the one that I adore
creed of my silent suffocation
Break this bittersweet spell on me
lost in the arms of destiny'''

        lyrics2 = '''There is something in the way
You're always somewhere else
Feelings have deserted me
To a point of no return
I don't believe in God
But I pray for you'''

        vbox = wx.BoxSizer(wx.VERTICAL)
        panel = wx.Panel(self, -1)
        st1 = wx.StaticText(panel, -1, lyrics1, style=wx.ALIGN_CENTRE)
        st2 = wx.StaticText(panel, -1, lyrics2, style=wx.ALIGN_CENTRE)
        vbox.Add(st1, 1, wx.EXPAND |  wx.TOP | wx.BOTTOM, 15)
        vbox.Add(st2, 1, wx.EXPAND |  wx.TOP | wx.BOTTOM, 15)
        panel.SetSizer(vbox)
        self.Centre()
        self.Show(True)


app = wx.App()
StaticText(None, -1, 'statixtext.py')
app.MainLoop()
```

wx.StaticBox

This is a kind of a decorator widget. It is used to logically group various widgets. Note that this widget must be created before the widgets that it contains, and that those widgets should be siblings, not children, of the static box.

Figure: staticbox.py

```
#!/usr/bin/python

# staticbox.py

import wx

class StaticBox(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(250, 230))

        wx.StaticBox(self, -1, 'Personal Info', (5, 5), size=(240, 170))
        wx.CheckBox(self, -1 ,'Male', (15, 30))
        wx.CheckBox(self, -1 ,'Married', (15, 55))
        wx.StaticText(self, -1, 'Age', (15, 95))
        wx.SpinCtrl(self, -1, '1', (55, 90), (60, -1), min=1, max=120)
        wx.Button(self, 1, 'Ok', (90, 185), (60, -1))

        self.Bind(wx.EVT_BUTTON, self.OnClose, id=1)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnClose(self, event):
        self.Close()

app = wx.App()
StaticBox(None, -1, 'staticbox.py')
app.MainLoop()
```

wx.ComboBox

wx.ComboBox is a combination of a single line text field, a button with a down arrow image and a listbox. When you press the button, a listbox appears. User can select only one option from the supplied string list. wx.ComboBox has the following constructor:

```
wx.ComboBox(int id, string value='', wx.Point pos=wx.DefaultPosition, wx.S
   wx.List choices=wx.EmptyList, int style=0, wx.Validator validator=wx.De
   string name=wx.ComboBoxNameStr)
```

wx.ComboBox styles

- wx.CB_DROPDOWN
- wx.CB_READONLY
- wx.CB_SORT

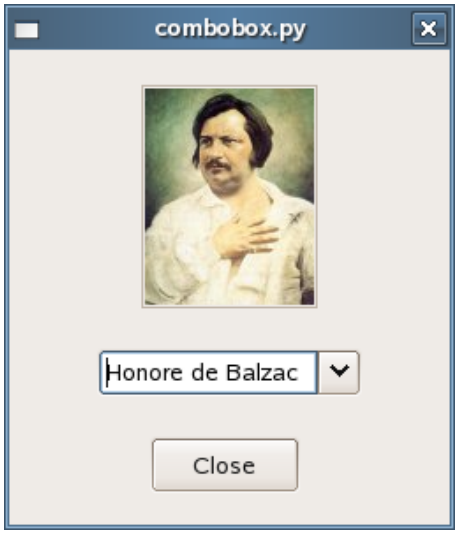


Figure: combobox.py

```
#!/usr/bin/python

# combobox.py

import wx

class ComboBox(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(250, 270))

        panel = wx.Panel(self, -1, (75, 20), (100, 127), style=wx.SUNKEN_
        self.picture = wx.StaticBitmap(panel)
        panel.SetBackgroundColour(wx.WHITE)

        self.images = ['tolstoy.jpg', 'feuchtwanger.jpg', 'balzac.jpg', '
                    'galsworthy.jpg', 'wolfe.jpg', 'zweig.jpg']
        authors = ['Leo Tolstoy', 'Lion Feuchtwanger', 'Honore de Balzac'
                'Boris Pasternak', 'John Galsworthy', 'Tom Wolfe', 'Stefa

        wx.ComboBox(self, -1, pos=(50, 170), size=(150, -1), choices=auth
                                        style=wx.CB_READONLY)
        wx.Button(self, 1, 'Close', (80, 220))
```

```
        self.Bind(wx.EVT_BUTTON, self.OnClose, id=1)
        self.Bind(wx.EVT_COMBOBOX, self.OnSelect)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnClose(self, event):
        self.Close()

    def OnSelect(self, event):
        item = event.GetSelection()
        self.picture.SetFocus()
        self.picture.SetBitmap(wx.Bitmap('images/' + self.images[item]))


app = wx.App()
ComboBox(None, -1, 'combobox.py')
app.MainLoop()
```

wx.CheckBox

wx.CheckBox is a widget that has two states. On and Off. It is a box with a label. The label can be set to the right or to the left of the box. If the checkbox is checked, it is represented by a tick in a box. wx.CheckBox Styles

- wx.ALIGN_RIGHT

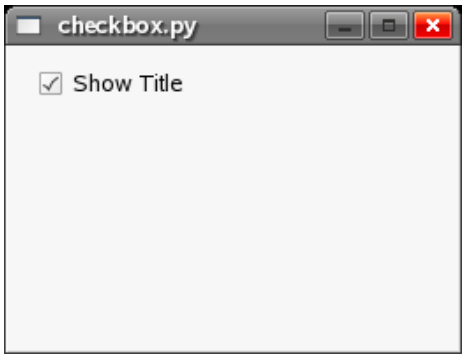


Figure: checkbox.py

```
#!/usr/bin/python

# checkbox.py

import wx

class CheckBox(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 170))

        panel = wx.Panel(self, -1)
```

```
        self.cb = wx.CheckBox(panel, -1, 'Show Title', (10, 10))
        self.cb.SetValue(True)

        wx.EVT_CHECKBOX(self, self.cb.GetId(), self.ShowTitle)

        self.Show()
        self.Centre()

    def ShowTitle(self, event):
        if self.cb.GetValue():
            self.SetTitle('checkbox.py')
        else: self.SetTitle('')


app = wx.App()
CheckBox(None, -1, 'checkbox.py')
app.MainLoop()
```

wx.StatusBar

As it's name indicates, the wx.StatusBar widget is used to display application status information. It can be divided into several parts to show different kind of information. We can insert other widgets into the wx.StatusBar. It can be used as an alternative to dialogs, since dialogs are ofted abused and they are disliked by most users. We can create a wx.StatusBar in two ways. We can manually create our own wx.StatusBar and call SetStatusBar() method or we can simply call the CreateStatusBar() method. The latter method creates a default wx.StatusBar for us. In our example, we have a wx.Frame widget and five other widgets. If we hover a mouse pointer over a widget, it's description is shown on the wx.StatusBar

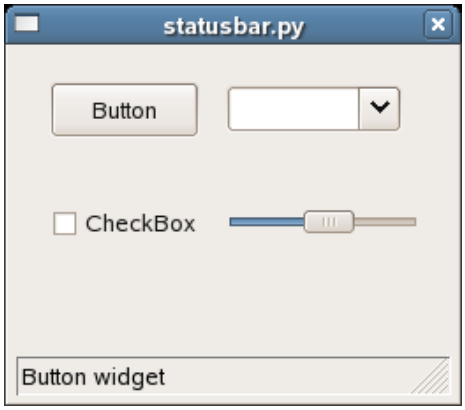


Figure: statusbar.py

```
#!/usr/bin/python

# statusbar.py
```

```
import wx

class Statusbar(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 200),
        style=wx.CAPTION | wx.SYSTEM_MENU | wx.CLOSE_BOX)

        panel = wx.Panel(self, 1)

        button = wx.Button(panel, 2, 'Button', (20, 20))
        text = wx.CheckBox(panel, 3, 'CheckBox', (20, 90))
        combo = wx.ComboBox(panel, 4, '', (120, 22))
        slider = wx.Slider(panel, 5, 6, 1, 10, (120, 90), (110, -1))

        panel.Bind(wx.EVT_ENTER_WINDOW, self.EnterPanel, id=1)
        button.Bind(wx.EVT_ENTER_WINDOW, self.EnterButton, id=2)
        text.Bind(wx.EVT_ENTER_WINDOW, self.EnterText, id=3)
        combo.Bind(wx.EVT_ENTER_WINDOW, self.EnterCombo, id=4)
        slider.Bind(wx.EVT_ENTER_WINDOW, self.EnterSlider, id=5)

        self.sb = self.CreateStatusBar()
        self.SetMaxSize((250, 200))
        self.SetMinSize((250, 200))
        self.Show(True)
        self.Centre()

    def EnterButton(self, event):
        self.sb.SetStatusText('Button widget')
        event.Skip()

    def EnterPanel(self, event):
        self.sb.SetStatusText('Panel widget')
        event.Skip()

    def EnterText(self, event):
        self.sb.SetStatusText('CheckBox widget')
        event.Skip()

    def EnterCombo(self, event):
        self.sb.SetStatusText('ComboBox widget')
        event.Skip()

    def EnterSlider(self, event):
        self.sb.SetStatusText('Slider widget')
        event.Skip()

app = wx.App()
Statusbar(None, -1, 'statusbar.py')
app.MainLoop()
```

wx.RadioButton

wx.RadioButton is a widget that allows the user to select a single exclusive choice from a group of options. A group of radio buttons is defined by having the first RadioButton in the group contain the wx.RB_GROUP style. All other RadioButtons defined after the first RadioButton with this style flag is set will be added to the function

group of the first RadioButton. Declaring another RadioButton with the wx.RB_GROUP flag will start a new radio button group.

wx.RadioButton Styles

- wx.RB_GROUP
- wx.RB_SINGLE
- wx.CB_SORT

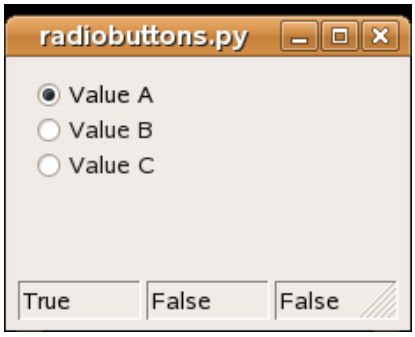


Figure: radiobutton.py

```
#!/usr/bin/python

# radiobuttons.py

import wx

class RadioButtons(wx.Frame):
    def __init__(self, parent, id, title, size=(210, 150)):
        wx.Frame.__init__(self, parent, id, title)
        panel = wx.Panel(self, -1)
        self.rb1 = wx.RadioButton(panel, -1, 'Value A', (10, 10), style=w
        self.rb2 = wx.RadioButton(panel, -1, 'Value B', (10, 30))
        self.rb3 = wx.RadioButton(panel, -1, 'Value C', (10, 50))

        self.Bind(wx.EVT_RADIOBUTTON, self.SetVal, id=self.rb1.GetId())
        self.Bind(wx.EVT_RADIOBUTTON, self.SetVal, id=self.rb2.GetId())
        self.Bind(wx.EVT_RADIOBUTTON, self.SetVal, id=self.rb3.GetId())

        self.statusbar = self.CreateStatusBar(3)
        self.SetVal(True)
        self.Centre()
        self.Show(True)

    def SetVal(self, event):
        state1 = str(self.rb1.GetValue())
        state2 = str(self.rb2.GetValue())
        state3 = str(self.rb3.GetValue())

        self.statusbar.SetStatusText(state1, 0)
        self.statusbar.SetStatusText(state2, 1)
        self.statusbar.SetStatusText(state3, 2)


app = wx.App()
```

```
RadioButtons(None, -1, 'radiobuttons.py')
app.MainLoop()
```

wx.Gauge

wx.Gauge is a widget that is used, when we process lengthy tasks.

wx.Gauge styles

- wx.GA_HORIZONTAL
- wx.GA_VERTICAL

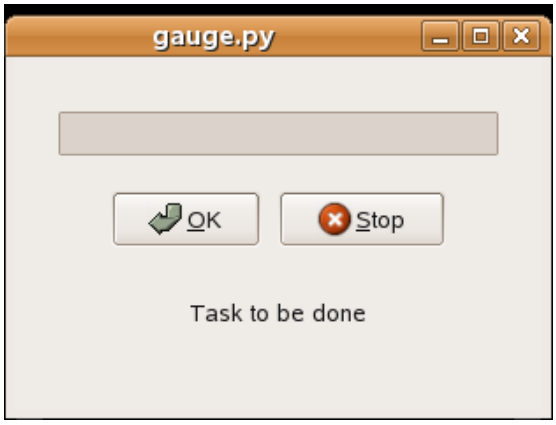


Figure: gauge.py

```
# gauge.py

import wx

class Gauge(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 200))

        self.timer = wx.Timer(self, 1)
        self.count = 0

        self.Bind(wx.EVT_TIMER, self.OnTimer, self.timer)

        panel = wx.Panel(self, -1)
        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox1 = wx.BoxSizer(wx.HORIZONTAL)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)
        hbox3 = wx.BoxSizer(wx.HORIZONTAL)

        self.gauge = wx.Gauge(panel, -1, 50, size=(250, 25))
        self.btn1 = wx.Button(panel, wx.ID_OK)
        self.btn2 = wx.Button(panel, wx.ID_STOP)
        self.text = wx.StaticText(panel, -1, 'Task to be done')

        self.Bind(wx.EVT_BUTTON, self.OnOk, self.btn1)
        self.Bind(wx.EVT_BUTTON, self.OnStop, self.btn2)
```

```
        hbox1.Add(self.gauge, 1, wx.ALIGN_CENTRE)
        hbox2.Add(self.btn1, 1, wx.RIGHT, 10)
        hbox2.Add(self.btn2, 1)
        hbox3.Add(self.text, 1)
        vbox.Add((0, 30), 0)
        vbox.Add(hbox1, 0, wx.ALIGN_CENTRE)
        vbox.Add((0, 20), 0)
        vbox.Add(hbox2, 1, wx.ALIGN_CENTRE)
        vbox.Add(hbox3, 1, wx.ALIGN_CENTRE)

        panel.SetSizer(vbox)
        self.Centre()
        self.Show(True)

    def OnOk(self, event):
        if self.count >= 50:
            return
        self.timer.Start(100)
        self.text.SetLabel('Task in Progress')

    def OnStop(self, event):
        if self.count == 0 or self.count >= 50 or not self.timer.IsRunnir
            return
        self.timer.Stop()
            self.text.SetLabel('Task Interrupted')
        wx.Bell()

    def OnTimer(self, event):
        self.count = self.count +1
        self.gauge.SetValue(self.count)
        if self.count == 50:
            self.timer.Stop()
            self.text.SetLabel('Task Completed')

app = wx.App()
Gauge(None, -1, 'gauge.py')
app.MainLoop()
```

wx.Slider

wx.Slider is a widget that has a simple handle. This handle can be pulled back and forth. This way we are choosing a value for a specific task. Say we want to input into our application the age of a customer. For this purpose, wx.Slider might be a good choice.
wx.Slider styles

- wxSL_HORIZONTAL
- wxSL_VERTICAL
- wxSL_AUTOTICKS
- wxSL_LABELS
- wxSL_LEFT
- wxSL_RIGHT
- wxSL_TOP
- wxSL_BOTTOM
- wxSL_INVERSE

Figure: slider.py

```
#!/usr/bin/python

# slider.py

import wx

class Slider(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 150))

        panel = wx.Panel(self, -1)
        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        self.sld = wx.Slider(panel, -1, 200, 150, 500, (-1, -1), (250, -1
                wx.SL_AUTOTICKS | wx.SL_HORIZONTAL | wx.SL_LABELS)
        btn1 = wx.Button(panel, 1, 'Adjust')
        btn2 = wx.Button(panel, 2,  'Close')

        wx.EVT_BUTTON(self, 1, self.OnOk)
        wx.EVT_BUTTON(self, 2, self.OnClose)

        vbox.Add(self.sld, 1, wx.ALIGN_CENTRE)
        hbox.Add(btn1, 1, wx.RIGHT, 10)
        hbox.Add(btn2, 1)
        vbox.Add(hbox, 0, wx.ALIGN_CENTRE | wx.ALL, 20)
        panel.SetSizer(vbox)
        self.Centre()
        self.Show(True)

    def OnOk(self, event):
        val = self.sld.GetValue()
        self.SetSize((val*2, val))

    def OnClose(self, event):
        self.Close()

app = wx.App()
Slider(None, -1, 'slider.py')
app.MainLoop()
```

wx.ListBox

wx.Listbox is a widget that consists of a scrolling box and a list of items. User can select one or more items from that list. It depends on whether it is created as a single or multiple selection box. Selected items are marked.

listbox.py example consists of four different widgets. wx.Listbox, wx.TextCtrl, wx.StaticText and wx.Button. Widgets are organized with sizer-s. wx.Listbox has a list of six different world times. These abbreviations are explained in the second wx.TextCtrl. Current time is displayed in the wx.StaticText widget. wx.Timer widget is used to update the time every 100 miliseconds.

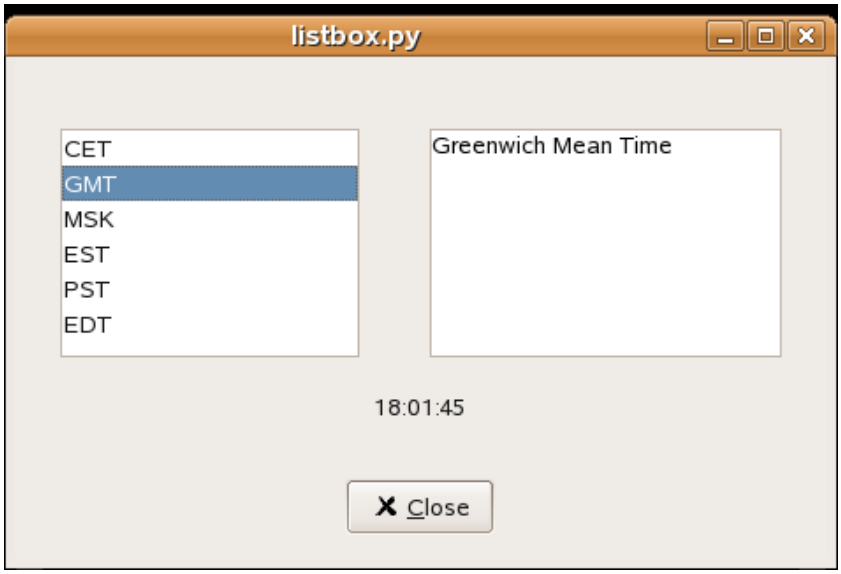


Figure: listbox.py

```
#!/usr/bin/python

# listbox.py

import wx
from time import *

class Listbox(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(550, 350))

        zone_list = ['CET', 'GMT', 'MSK', 'EST', 'PST', 'EDT']

        self.full_list = {
            'CET': 'Central European Time',
            'GMT': 'Greenwich Mean Time',
            'MSK': 'Moscow Time',
            'EST': 'Eastern Standard Time',
            'PST': 'Pacific Standard Time',
            'EDT': 'Eastern Daylight Time'
```

```
        }

        self.time_diff = {
            'CET' : 1,
            'GMT' : 0,
            'MSK': 3,
            'EST': -5,
            'PST': -8,
            'EDT': -4
        }

        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox1 = wx.BoxSizer(wx.HORIZONTAL)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)
        hbox3 = wx.BoxSizer(wx.HORIZONTAL)

        self.timer = wx.Timer(self, 1)
        self.diff = 0

        panel = wx.Panel(self, -1)

        self.time_zones = wx.ListBox(panel, 26, (-1, -1), (170, 130),
                zone_list, wx.LB_SINGLE)
        self.time_zones.SetSelection(0)
        self.text = wx.TextCtrl(panel, -1, 'Central European Time',
                size=(200, 130), style=wx.TE_MULTILINE)
        self.time = wx.StaticText(panel, -1, '')
        btn = wx.Button(panel, wx.ID_CLOSE, 'Close')
        hbox1.Add(self.time_zones, 0, wx.TOP, 40)
        hbox1.Add(self.text, 1, wx.LEFT | wx.TOP, 40)
        hbox2.Add(self.time, 1, wx.ALIGN_CENTRE)
        hbox3.Add(btn, 0, wx.ALIGN_CENTRE)
        vbox.Add(hbox1, 0, wx.ALIGN_CENTRE)
        vbox.Add(hbox2, 1, wx.ALIGN_CENTRE)
        vbox.Add(hbox3, 1, wx.ALIGN_CENTRE)

        panel.SetSizer(vbox)

        self.timer.Start(100)

        wx.EVT_BUTTON(self, wx.ID_CLOSE, self.OnClose)
        wx.EVT_LISTBOX(self, 26, self.OnSelect)
        wx.EVT_TIMER(self, 1, self.OnTimer)

        self.Show(True)
        self.Centre()

    def OnClose(self, event):
        self.Close()

    def OnSelect(self, event):
        index = event.GetSelection()
        time_zone = self.time_zones.GetString(index)
        self.diff = self.time_diff[time_zone]
        self.text.SetValue(self.full_list[time_zone])

    def OnTimer(self, event):
        ct = gmtime()
        print_time = (ct[0], ct[1], ct[2], ct[3]+self.diff,
                      ct[4], ct[5], ct[6], ct[7], -1)
```

```
        self.time.SetLabel(strftime("%H:%M:%S", print_time))


app = wx.App()
Listbox(None, -1, 'listbox.py')
app.MainLoop()
```

wx.SpinCtrl

This widget lets you increment and decrement a value. It has two up and down arrow buttons for this purpose. User can enter a value into a box or increment/decrement it by these two arrows. Converter script converts Fahrenheit temperature to Celsius. This example is very popular and can be found in most programming primer books. So I made a wxPython example as well. wx.SpinCtrl styles

- wx.SP_ARROW_KEYS
- wx.SP_WRAP

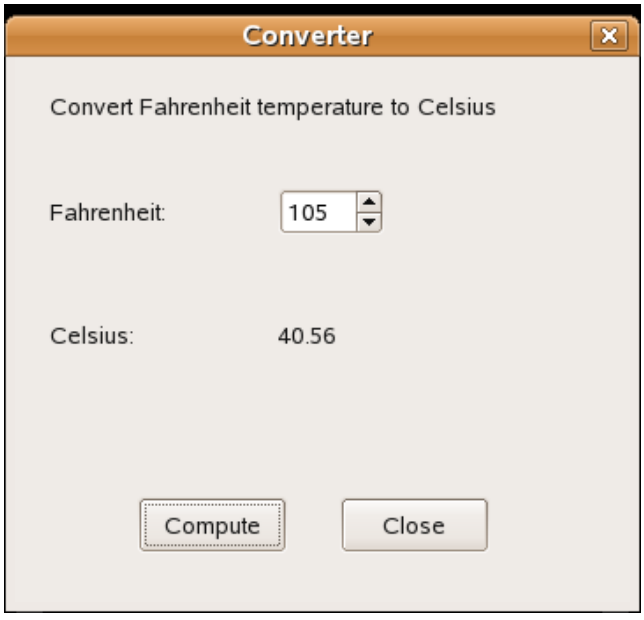


Figure: converter.py

```
#!/usr/bin/python

# spinctrl.py

import wx

class Converter(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(350, 310))
```

```
        wx.StaticText(self, -1, 'Convert Fahrenheit temperature to Celsiu
        wx.StaticText(self, -1, 'Fahrenheit: ', (20, 80))
        wx.StaticText(self, -1, 'Celsius: ', (20, 150))
        self.celsius =  wx.StaticText(self, -1, '', (150, 150))
        self.sc = wx.SpinCtrl(self, -1, '',  (150, 75), (60, -1))
        self.sc.SetRange(-459, 1000)
        self.sc.SetValue(0)
        compute_btn = wx.Button(self, 1, 'Compute', (70, 250))
        compute_btn.SetFocus()
        clear_btn = wx.Button(self, 2, 'Close', (185, 250))

        wx.EVT_BUTTON(self, 1, self.OnCompute)
        wx.EVT_BUTTON(self, 2, self.OnClose)
        wx.EVT_CLOSE(self, self.OnClose)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnCompute(self, event):
        fahr = self.sc.GetValue()
        cels = round((fahr-32)*5/9.0, 2)
        self.celsius.SetLabel(str(cels))

    def OnClose(self, event):
        self.Destroy()

app = wx.App()
Converter(None, -1, 'Converter')
app.MainLoop()
```

wx.SplitterWindow

This widget enables to split the main area of an application into parts. The user can dynamically resize those parts with the mouse pointer. Such a solution can be seen in mail clients (evolution) or in burning software (k3b). You can split an area vertically or horizontally.

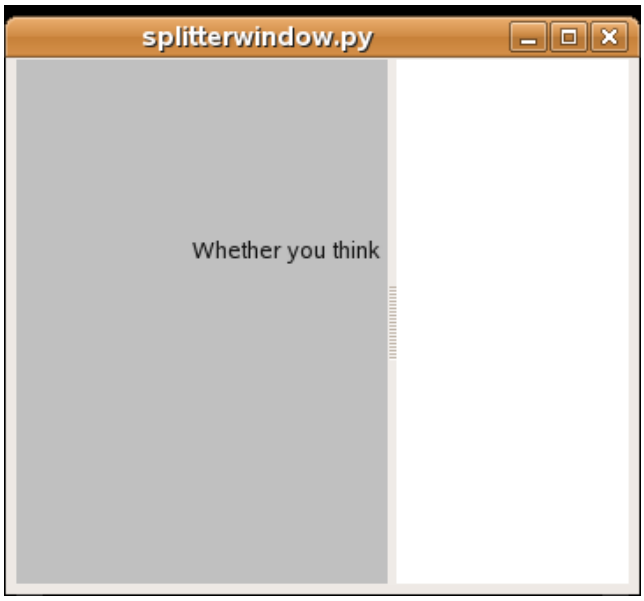


Figure: splitterwindow.py

```
#!/usr/bin/python

# splitterwindow.py

import wx

class Splitterwindow(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 300))

        quote = '''Whether you think that you can, or that you can't, you
usually right'''

        splitter = wx.SplitterWindow(self, -1)
        panel1 = wx.Panel(splitter, -1)
        wx.StaticText(panel1, -1, quote, (100, 100), style=wx.ALIGN_CENTR

        panel1.SetBackgroundColour(wx.LIGHT_GREY)
        panel2 = wx.Panel(splitter, -1)
        panel2.SetBackgroundColour(wx.WHITE)
        splitter.SplitVertically(panel1, panel2)
        self.Centre()
        self.Show(True)

app = wx.App()
Splitterwindow(None, -1, 'splitterwindow.py')
app.MainLoop()
```

wx.ScrolledWindow

This is one of the container widgets. It can be useful, when we have a larger area than a window can display. In our example, we demonstrate such a case. We place a large image into our window. When the window is smaller than our image, Scrollbars are

displayed automatically.

Figure: scrolledwindow.py

```
#!/usr/bin/python

# scrolledwindow.py

import wx

class ScrolledWindow(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(500, 400))

        sw = wx.ScrolledWindow(self)
        bmp = wx.Image('images/aliens.jpg',wx.BITMAP_TYPE_JPEG).ConvertTo
        wx.StaticBitmap(sw, -1, bmp)
        sw.SetScrollbars(20, 20, 55, 40)
        sw.Scroll(50,10)
        self.Centre()
        self.Show()

app = wx.App()
ScrolledWindow(None, -1, 'Aliens')
app.MainLoop()
```

The SetScrollbars() method creates horizontal and vertical scrollbars. By calling the Scroll() method we programmatically scroll to the given position.

wx.Notebook

wx.Notebook widget joins multiple windows with corresponding tabs. You can position the Notebook widget using the following style flags:

- wx.NB_LEFT
- wx.NB_RIGHT
- wx.NB_TOP
- wx.NB_BOTTOM

The default position is wx.NB_TOP.

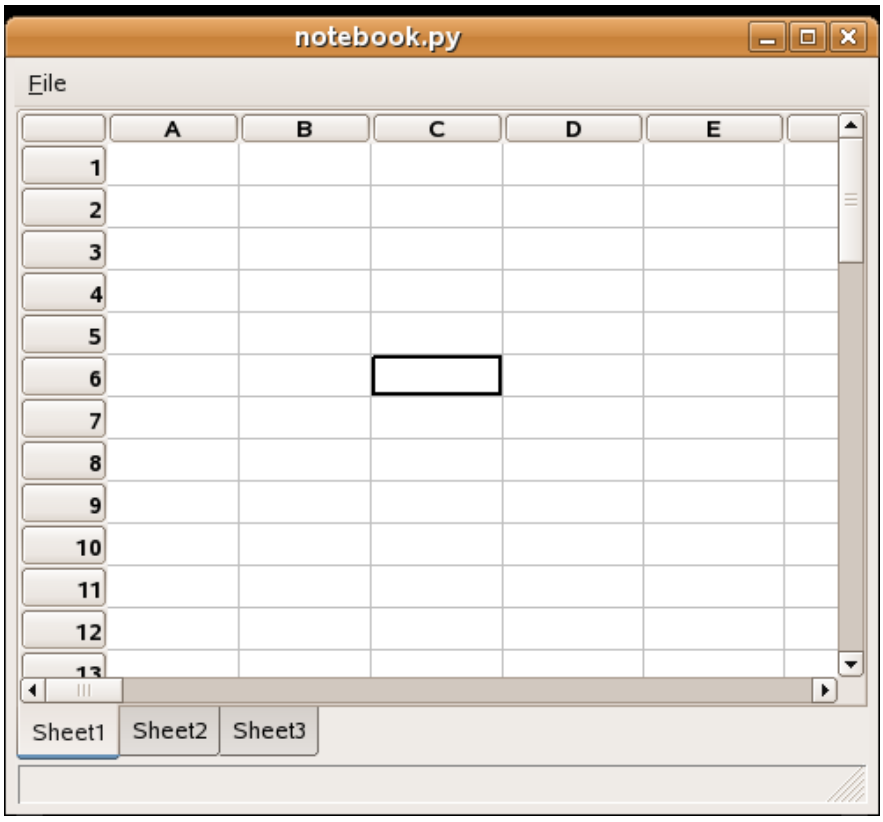


Figure: notebook.py

```
#!/usr/bin/python

# notebook.py

import wx
import wx.lib.sheet as sheet
```

```
class MySheet(sheet.CSheet):
    def __init__(self, parent):
        sheet.CSheet.__init__(self, parent)
        self.SetNumberRows(50)
        self.SetNumberCols(50)

class Notebook(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(600, 500))
        menubar = wx.MenuBar()
        file = wx.Menu()
        file.Append(101, 'Quit', '' )
        menubar.Append(file, '&File')
        self.SetMenuBar(menubar)

        wx.EVT_MENU(self, 101, self.OnQuit)

        nb = wx.Notebook(self, -1, style=wx.NB_BOTTOM)
        self.sheet1 = MySheet(nb)
        self.sheet2 = MySheet(nb)
        self.sheet3 = MySheet(nb)

        nb.AddPage(self.sheet1, 'Sheet1')
        nb.AddPage(self.sheet2, 'Sheet2')
        nb.AddPage(self.sheet3, 'Sheet3')

        self.sheet1.SetFocus()
        self.StatusBar()
        self.Centre()
        self.Show()

    def StatusBar(self):
        self.statusbar = self.CreateStatusBar()

    def OnQuit(self, event):
        self.Close()

app = wx.App()
Notebook(None, -1, 'notebook.py')
app.MainLoop()
```

In our example we create a notebook widget with wx.NB_BOTTOM style. It is therefore positioned on the bottom of the frame accordingly. We add various widgets into the notebook widget with the AddPage() method. We put simple spreadsheet widgets. A Spreadsheet widget can be foung in wx.lib.sheet module.

wx.Panel

wx.Panel is a basic parent widget. It adds some basic functionality to the *wx.Window* widget, which is usually not used directly. Normally we create a *wx.Frame* widget first. Then we place a

wx.Panel widget inside this frame. Then we place widgets on the panel. This is the common scenario. However, we can also combine panels to create interesting interface. In the following example we create a two side window with headers. We use altogether six different *wx.Panel* widgets.

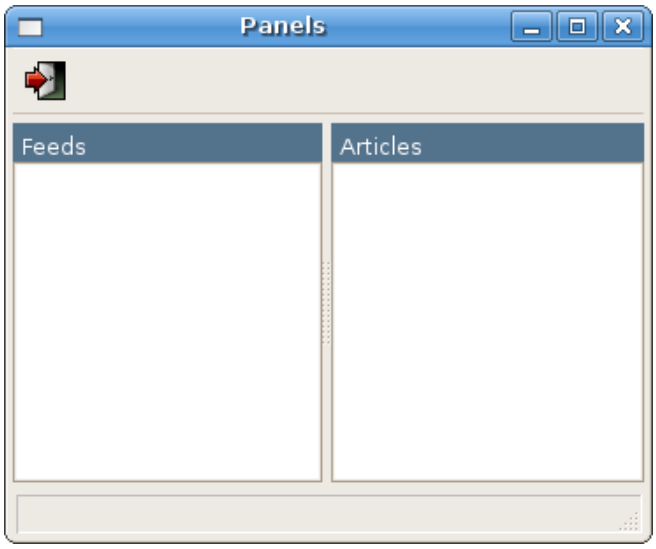


Figure: panels.py

```
#!/usr/bin/python

# panels.py

import wx


class Panels(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        splitter = wx.SplitterWindow(self, -1)

        vbox1 = wx.BoxSizer(wx.VERTICAL)
        panel1 = wx.Panel(splitter, -1)
        panel11 = wx.Panel(panel1, -1, size=(-1, 40))
        panel11.SetBackgroundColour('#53728c')
        st1 = wx.StaticText(panel11, -1, 'Feeds', (5, 5))
        st1.SetForegroundColour('WHITE')

        panel12 = wx.Panel(panel1, -1, style=wx.BORDER_SUNKEN)
        panel12.SetBackgroundColour('WHITE')

        vbox1.Add(panel11, 0, wx.EXPAND)
        vbox1.Add(panel12, 1, wx.EXPAND)

        panel1.SetSizer(vbox1)

        vbox2 = wx.BoxSizer(wx.VERTICAL)
```

```
        panel2 = wx.Panel(splitter, -1)
        panel21 = wx.Panel(panel2, -1, size=(-1, 40), style=wx.NO_BORDER)
        st2 = wx.StaticText(panel21, -1, 'Articles', (5, 5))
        st2.SetForegroundColour('WHITE')

        panel21.SetBackgroundColour('#53728c')
        panel22 = wx.Panel(panel2, -1, style=wx.BORDER_RAISED)

        panel22.SetBackgroundColour('WHITE')
        vbox2.Add(panel21, 0, wx.EXPAND)
        vbox2.Add(panel22, 1, wx.EXPAND)

        panel2.SetSizer(vbox2)

        toolbar = self.CreateToolBar()
        toolbar.AddLabelTool(1, 'Exit', wx.Bitmap('icons/stock_exit.png')
        toolbar.Realize()
        self.Bind(wx.EVT_TOOL, self.ExitApp, id=1)

        hbox.Add(splitter, 1, wx.EXPAND | wx.TOP | wx.BOTTOM, 5)
        self.SetSizer(hbox)
        self.CreateStatusBar()
        splitter.SplitVertically(panel1, panel2)
        self.Centre()
        self.Show(True)


    def ExitApp(self, event):
        self.Close()


app = wx.App()
Panels(None, -1, 'Panels')
app.MainLoop()
```

```
hbox = wx.BoxSizer(wx.HORIZONTAL)
splitter = wx.SplitterWindow(self, -1)
```

The *wx.SplitterWindow* will split our window into two parts.

One panel will be placed on the left and one on the right side. Each one will have other two panels. One will create a header and the other one will take up the rest of the parent panel. Together we will use six panels.

```
panel11 = wx.Panel(panel1, -1, size=(-1, 40))
panel11.SetBackgroundColour('#53728c')
st1 = wx.StaticText(panel11, -1, 'Feeds', (5, 5))
st1.SetForegroundColour('WHITE')
...
vbox1.Add(panel11, 0, wx.EXPAND)
```

Here we create the header panel. The header height is 40px. The

color is set to dark blue. (#53728c) We put a *wx.StaticText* inside the header panel. The position is 5px from left and 5px from top so that we have some space between the panel and the static text. The color of the static text is set to white. In the end, we make the panel11 expandable and set the proportion to 0.

```
panel12 = wx.Panel(panel1, -1, style=wx.BORDER_SUNKEN)
panel12.SetBackgroundColour('WHITE')
...
vbox1.Add(panel12, 1, wx.EXPAND)
```

The bottom panel is created with *wx.BORDER_SUNKEN* style. The color is set to white. We make it expandable and set the proportion parameter to 1.

Home ‡ Contents ‡ Top of Page

Advanced widgets in wxPython

In the following chapters we will talk about advanced widgets. A big advantage of wxPython over a competing PyGTK is the availability of a huge amount of advanced widgets. PyGTK is a layer over a C based GKT+ toolkit. It does not provide new widgets. In contrast, wxPython is a layer over wxWidgets a C++ based toolkit. wxWidgets consists of a large group of widgets. All this widgets are created in C++. wxPython is a glue that combines python language with this toolkit. If we want to have a grid widget in our application using PyGTK, we have to create it ourselves. Such a widget is quite complicated. Not to mention the speed penalty. Dynamic languages like Python, PERL or Ruby are not suitable for such tasks. Dynamic languages are great in various areas. They are simple to use. They are great for prototyping, in house developing or for studying computer programming. If we need a quick solution or we need an application, that will change rapidly over a short period of time, dynamic languages are superior to compiled languages. On the other hand, if we develop resource intensive applications, games, high quality multimedia applications, there is no competition to C++.

wxPython has several well known advanced widgets. For example a tree widget, an html window, a grid widget, a listbox widget, a list widget or an editor with advanced styling capabilities. wxPython and wxWidgets are being developed all the time. New widgets and features emerge with every major release. At the time when I write these words a wxPython 2.8.3.0 has been released just two days ago. (22-Mar-2007).

A wx.ListBox widget

A *wx.ListBox* widget is used for displaying and working with a list of items. As it's name indicates, it is a rectangle that has a list of strings inside. We could use it for displaying a list of mp3 files, book names, module names of a larger project or names of our friends. A *wx.ListBox* can be created in two different states. In a single selection state or a multiple selection state. The single selection state is the default state. There are two significant events in *wx.ListBox*. The first one is the *wx.EVT_COMMAND_LISTBOX_SELECTED* event. This event is generated when we select a string in a *wx.ListBox*. The second one is the *wx.EVT_COMMAND_LISTBOX_DOUBLE_CLICKED* event. It is generated when we double click an item in a *wx.ListBox*. The number of elements inside a wx.ListBox is limited on GTK platform. According to the documentation, it is currently around 2000 elements. Quite enough, I think. The elements are numbered from zero. Scrollbars are displayed automatically if needed.

The constructor of a wx.ListBox widget is as follows:

```
wx.ListBox(wx.Window parent, int id=-1, wx.Point pos=wx.DefaultPosition, wx.Size size=
       list choices=[], long style=0, wx.Validator validator=wx.DefaultValidator,
       string name=wx.ListBoxNameStr)
```

There is a choices parameter. If we put some values there, they will be displayed from the construction of the widget. This parameter is empty by default.

In our code example we have a listbox and four buttons. Each of them calls a different method of our listbox. If we want to append a new item, we call the *Append()* method. If we want to delete an item, we call the *Delete()* method. To clear all strings in a listbox, we call the Clear() method.

```
#!/usr/bin/python

# listbox.py

import wx

ID_NEW = 1
ID_RENAME = 2
ID_CLEAR = 3
ID_DELETE = 4


class ListBox(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 220))

        panel = wx.Panel(self, -1)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        self.listbox = wx.ListBox(panel, -1)
        hbox.Add(self.listbox, 1, wx.EXPAND | wx.ALL, 20)

        btnPanel = wx.Panel(panel, -1)
        vbox = wx.BoxSizer(wx.VERTICAL)
        new = wx.Button(btnPanel, ID_NEW, 'New', size=(90, 30))
        ren = wx.Button(btnPanel, ID_RENAME, 'Rename', size=(90, 30))
        dlt = wx.Button(btnPanel, ID_DELETE, 'Delete', size=(90, 30))
        clr = wx.Button(btnPanel, ID_CLEAR, 'Clear', size=(90, 30))

        self.Bind(wx.EVT_BUTTON, self.NewItem, id=ID_NEW)
        self.Bind(wx.EVT_BUTTON, self.OnRename, id=ID_RENAME)
        self.Bind(wx.EVT_BUTTON, self.OnDelete, id=ID_DELETE)
        self.Bind(wx.EVT_BUTTON, self.OnClear, id=ID_CLEAR)
        self.Bind(wx.EVT_LISTBOX_DCLICK, self.OnRename)

        vbox.Add((-1, 20))
        vbox.Add(new)
        vbox.Add(ren, 0, wx.TOP, 5)
        vbox.Add(dlt, 0, wx.TOP, 5)
        vbox.Add(clr, 0, wx.TOP, 5)

        btnPanel.SetSizer(vbox)
        hbox.Add(btnPanel, 0.6, wx.EXPAND | wx.RIGHT, 20)
        panel.SetSizer(hbox)
```

```
        self.Centre()
        self.Show(True)

    def NewItem(self, event):
        text = wx.GetTextFromUser('Enter a new item', 'Insert dialog')
        if text != '':
            self.listbox.Append(text)

    def OnRename(self, event):
        sel = self.listbox.GetSelection()
        text = self.listbox.GetString(sel)
        renamed = wx.GetTextFromUser('Rename item', 'Rename dialog', text)
        if renamed != '':
            self.listbox.Delete(sel)
            self.listbox.Insert(renamed, sel)


    def OnDelete(self, event):
        sel = self.listbox.GetSelection()
        if sel != -1:
            self.listbox.Delete(sel)

    def OnClear(self, event):
        self.listbox.Clear()


app = wx.App()
ListBox(None, -1, 'ListBox')
app.MainLoop()
```

```
self.listbox = wx.ListBox(panel, -1)
hbox.Add(self.listbox, 1, wx.EXPAND | wx.ALL, 20)
```

We create an empty *wx.ListBox*. We put a 20px border around the listbox.

```
self.Bind(wx.EVT_LISTBOX_DCLICK, self.OnRename)
```

We bind a *wx.EVT_COMMAND_LISTBOX_DOUBLE_CLICKED* event type with the *OnRename()* method using the *wx.EVT_LISTBOX_DCLICK* event binder. This way we show a rename dialog if we double click on a specific element in the listbox.

```
def NewItem(self, event):
    text = wx.GetTextFromUser('Enter a new item', 'Insert dialog')
    if text != '':
        self.listbox.Append(text)
```

We call the *NewItem()* method by clicking on the New button. This method shows a *wx.GetTextFromUser* dialog window. The text that we enter is returned to the text variable. If the text is not empty, we append it to the listbox with the *Append()* method.

```
def OnDelete(self, event):
    sel = self.listbox.GetSelection()
    if sel != -1:
        self.listbox.Delete(sel)
```

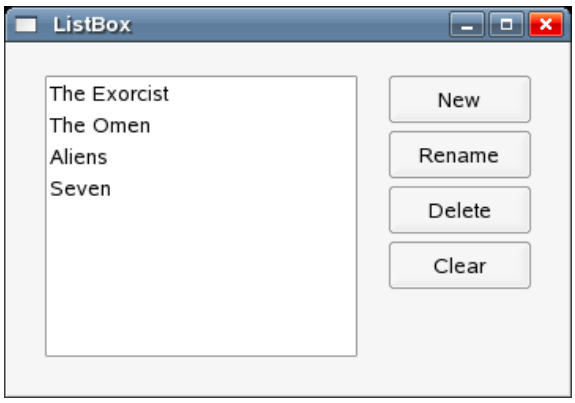
Deleting an item is done in two steps. First we find the index of the selected item by calling the *GetSelection()* method. Then we delete the item with the *Delete()* method. The parametor to the *Delete()* method is the selected index.

```
self.listbox.Delete(sel)
self.listbox.Insert(renamed, sel)
```

Notice, how we managed to rename a string. *wx.ListBox* widget has no Rename() method. We did this functionality by deleting the previously selected string and inserting a new string into the predecessor's position.

```
def OnClear(self, event):
    self.listbox.Clear()
```

The easiest thing is to clear the whole listbox. We simply call the *Clear()* method.



A wx.ListBox widget

A WX.HTML.HTMLWINDOW WIDGET

The *wx.html.HtmlWindow* widget displays html pages. It is not a full-fledged browser. We can do interesting things with *wx.html.HtmlWindow* widget.

Special formatting

For example in the following script we will create a window, that will display basic statistics. This formatting would be very hard if possible to create without *wx.html.HtmlWindow* widget.

```
#!/usr/bin/python

import wx
import wx.html as html

ID_CLOSE = 1
```

```
page = '<html><body bgcolor="#8e8e95"><table cellspacing="5" border="0" width="250">
<tr width="200" align="left"> \
<td bgcolor="#e7e7e7">  Maximum</td> \
<td bgcolor="#aaaaaa">  <b>9000</b></td> \
</tr> \
<tr align="left"> \
<td bgcolor="#e7e7e7">  Mean</td> \
<td bgcolor="#aaaaaa">  <b>6076</b></td> \
</tr> \
<tr align="left"> \
<td bgcolor="#e7e7e7">  Minimum</td> \
<td bgcolor="#aaaaaa">  <b>3800</b></td> \
</tr> \
<tr align="left"> \
<td bgcolor="#e7e7e7">  Median</td> \
<td bgcolor="#aaaaaa">  <b>6000</b></td> \
</tr> \
<tr align="left"> \
<td bgcolor="#e7e7e7">  Standard Deviation</td> \
<td bgcolor="#aaaaaa">  <b>6076</b></td> \
</tr> \
</body></table></html>'


class MyFrame(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(400, 290))

        panel = wx.Panel(self, -1)

        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        htmlwin = html.HtmlWindow(panel, -1, style=wx.NO_BORDER)
        htmlwin.SetBackgroundColour(wx.RED)
        htmlwin.SetStandardFonts()
        htmlwin.SetPage(page)

        vbox.Add((-1, 10), 0)
        vbox.Add(htmlwin, 1, wx.EXPAND | wx.ALL, 9)

        bitmap = wx.StaticBitmap(panel, -1, wx.Bitmap('images/newt.png'))
        hbox.Add(bitmap, 1, wx.LEFT | wx.BOTTOM | wx.TOP, 10)
        buttonOk = wx.Button(panel, ID_CLOSE, 'Ok')

        self.Bind(wx.EVT_BUTTON, self.OnClose, id=ID_CLOSE)

        hbox.Add((100, -1), 1, wx.EXPAND | wx.ALIGN_RIGHT)
        hbox.Add(buttonOk, flag=wx.TOP | wx.BOTTOM | wx.RIGHT, border=10)
        vbox.Add(hbox, 0, wx.EXPAND)

        panel.SetSizer(vbox)
        self.Centre()
        self.Show(True)

    def OnClose(self, event):
        self.Close()

app = wx.App(0)
MyFrame(None, -1, 'Basic Statistics')
app.MainLoop()
```

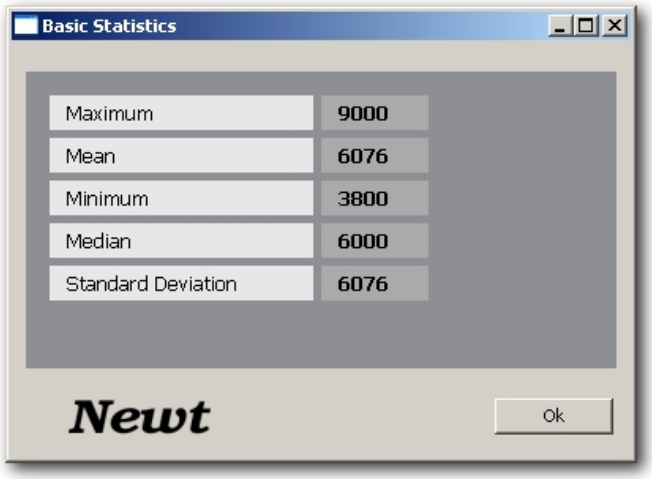


Figure: Html window example

Help window

We can use *wx.html.HtmlWindow* to provide help in our application. We can create a standalone window or we can create a window, that is going to be a part of the application. The following script will create a help window using the latter idea.

```
#!/usr/bin/python

# helpwindow.py

import wx
import wx.html as html

class HelpWindow(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(570, 400))

        toolbar = self.CreateToolBar()
        toolbar.AddLabelTool(1, 'Exit', wx.Bitmap('icons/exit.png'))
        toolbar.AddLabelTool(2, 'Help', wx.Bitmap('icons/help.png'))
        toolbar.Realize()

        self.splitter = wx.SplitterWindow(self, -1)
        self.panelLeft = wx.Panel(self.splitter, -1, style=wx.BORDER_SUNKEN)

        self.panelRight = wx.Panel(self.splitter, -1)
        vbox2 = wx.BoxSizer(wx.VERTICAL)
        header = wx.Panel(self.panelRight, -1, size=(-1, 20))
        header.SetBackgroundColour('#6f6a59')
        header.SetForegroundColour('WHITE')
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        st = wx.StaticText(header, -1, 'Help', (5, 5))
        font = st.GetFont()
        font.SetPointSize(9)
        st.SetFont(font)
        hbox.Add(st, 1, wx.TOP | wx.BOTTOM | wx.LEFT, 5)

        close = wx.BitmapButton(header, -1, wx.Bitmap('icons/fileclose.png', wx.BITMA
                style=wx.NO_BORDER)
        close.SetBackgroundColour('#6f6a59')
```

```
        hbox.Add(close, 0)
        header.SetSizer(hbox)

        vbox2.Add(header, 0, wx.EXPAND)

        help = html.HtmlWindow(self.panelRight, -1, style=wx.NO_BORDER)
        help.LoadPage('help.html')
        vbox2.Add(help, 1, wx.EXPAND)
        self.panelRight.SetSizer(vbox2)
        self.panelLeft.SetFocus()

        self.splitter.SplitVertically(self.panelLeft, self.panelRight)
        self.splitter.Unsplit()

        self.Bind(wx.EVT_BUTTON, self.CloseHelp, id=close.GetId())
        self.Bind(wx.EVT_TOOL, self.OnClose, id=1)
        self.Bind(wx.EVT_TOOL, self.OnHelp, id=2)

        self.Bind(wx.EVT_KEY_DOWN, self.OnKeyPressed)

        self.CreateStatusBar()

        self.Centre()
        self.Show(True)

    def OnClose(self, event):
        self.Close()

    def OnHelp(self, event):
        self.splitter.SplitVertically(self.panelLeft, self.panelRight)
        self.panelLeft.SetFocus()

    def CloseHelp(self, event):
        self.splitter.Unsplit()
        self.panelLeft.SetFocus()

    def OnKeyPressed(self, event):
        keycode = event.GetKeyCode()
        if keycode == wx.WXK_F1:
            self.splitter.SplitVertically(self.panelLeft, self.panelRight)
            self.panelLeft.SetFocus()


app = wx.App()
HelpWindow(None, -1, 'HelpWindow')
app.MainLoop()
```

The help window is hidden in the beginning. We can show it by clicking on the help button on the toolbar or by pressing F1. The help window appears on the right side of the application. To hide the help window, we click on the close button.

```
self.splitter.SplitVertically(self.panelLeft, self.panelRight)
self.splitter.Unsplit()
```

We create left a right panels and split them vertically. After that, we call the *Unsplit()* method. By default the method hides the right or bottom panes.

We divide the right panel into two parts. The header and the body of the panel. The header is an adjusted *wx.Panel*. The header consists of a static text and a bitmap button. We put *wx.html.Window* into the body of the panel.

```
close = wx.BitmapButton(header, -1, wx.Bitmap('icons/fileclose.png', wx.BITMAP_TYPE_PI
        style=wx.NO_BORDER)
close.SetBackgroundColour('#6f6a59')
```

The bitmap button style is set to *wx.NO_BORDER*. The background color is set to the color of the header panel. This is done in order to make the button appear as a part of the header.

```
help = html.HtmlWindow(self.panelRight, -1, style=wx.NO_BORDER)
help.LoadPage('help.html')
```

We create a wx.html.HtmlWindow widget on the right panel. We have our html code in a separate file. This time we call the *LoadPage()* method to obtain the html code.

```
self.panelLeft.SetFocus()
```

We set focus on the left panel. We can launch the help window with the F1 key. In order to control a window with a keyboard, it must have the focus. If we did not set the focus, we would have to first click on the panel and only then we could launch the help window with the F1 key press.

```
    def OnHelp(self, event):
        self.splitter.SplitVertically(self.panelLeft, self.panelRight)
        self.panelLeft.SetFocus()
```

To show the help window, we call the *OnHelp()* method. It splits the two panels vertically. We must not forget to set the focus again, because the initial focus is lost by splitting.

The following is the html file, that we load in our application.

```
<html>

<body bgcolor="#abababab">
<h4>Table of Contents</h4>

<ul>
<li><a href="#basic">Basic statistics</a></li>
<li><a href="#advanced">Advanced statistics</a></li>
<li><a href="#intro">Introducing Newt</a></li>
<li><a href="#charts">Working with charts</a></li>
<li><a href="#pred">Predicting values</a></li>
<li><a href="#neural">Neural networks</a></li>
<li><a href="#glos">Glossary</a></li>
</ul>

<p>
<a name="basic">
<h6>Basic Statistics</h6>
Overview of elementary concepts in statistics.
Variables. Correlation. Measurement scales. Statistical significance.
Distributions. Normality assumption.
</a>
```

```
</p>

<p>
<a name="advanced">
<h6>Advanced Statistics</h6>
Overview of advanced concepts in statistics. Anova. Linear regression.
Estimation and  hypothesis testing.
Error terms.
</a>
</p>

<p>
<a name="intro">
<h6>Introducing Newt</h6>
Introducing the basic functionality of the Newt application. Creating sheets.
Charts. Menus and Toolbars. Importing data. Saving data in various formats.
Exporting data. Shortcuts. List of methods.
</a>
</p>

<p>
<a name="charts">
<h6>Charts</h6>
Working with charts. 2D charts. 3D charts. Bar, line, box, pie, range charts.
Scatterplots. Histograms.
</a>
</p>

<p>
<a name="pred">
<h6>Predicting values</h6>
Time series and forecasting. Trend Analysis. Seasonality. Moving averages.
Univariate methods. Multivariate methods. Holt-Winters smoothing.
Exponential smoothing. ARIMA. Fourier analysis.
</a>
</p>

<p>
<a name="neural">
<h6>Neural networks</h6>
Overview of neural networks. Biology behind neural networks.
Basic artificial Model. Training. Preprocessing. Postprocessing.
Types of neural networks.
</a>
</p>

<p>
<a name="glos">
<h6>Glossary</h6>
Terms and definitions in statistics.
</a>
</p>

</body>
</html>
```

```
<li><a href="#basic">Basic statistics</a></li>
...
<a name="basic">
```

Normally I would write ... . Both are correct html notations. But *wx.html.HtmlWindow* supports only the first one. *wx.html.HtmlWindow* supports only a subset of the html markup language.

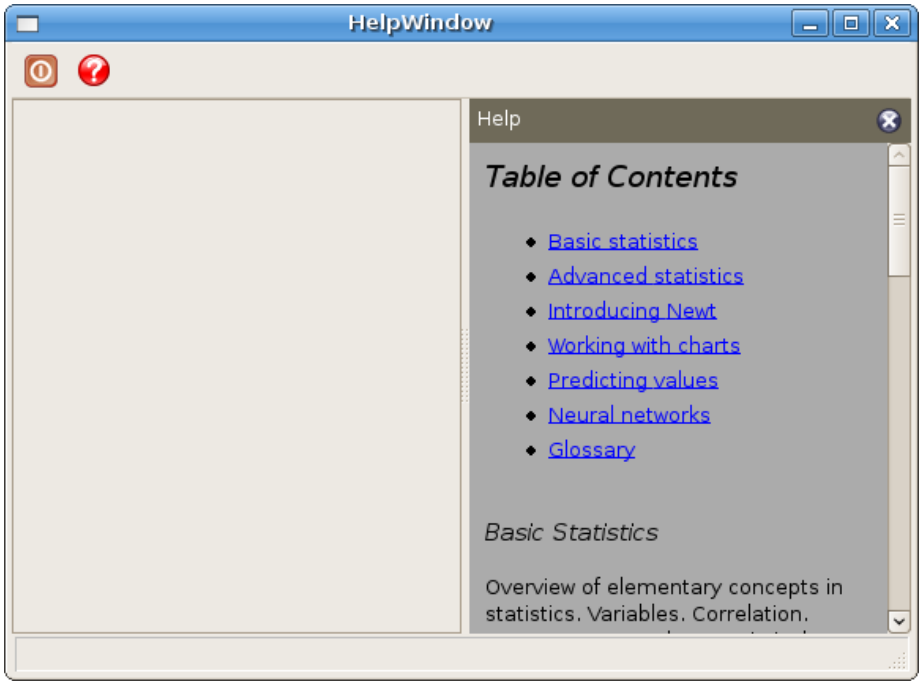


Figure: Help window

A WX.LISTCTRL WIDGET

A *wx.ListCtrl* is a graphical representation of a list of items. A *wx.ListBox* can only have one column. *wx.ListCtrl* can have more than one column. *wx.ListCtrl* is a very common and useful widget. For example a file manager uses a *wx.ListCtrl* to display directories and files on the file system. A cd burner application displays files to be burned inside a *wx.ListCtrl*.

A *wx.ListCtrl* can be used in three different formats. In a list view, report view or a icon view. These formats are controled by the *wx.ListCtrl* window styles. *wx.LC_REPORT*, *wx.LC_LIST* and *wx.LC_ICON*.

```
wx.ListCtrl(wx.Window parent, int id, wx.Point pos = (-1, -1), wx.Size size = (-1, -1)
int style = wx.LC_ICON, wx.Validator validator = wx.DefaultValidator, string name = wx
```

wx.ListCtrl styles

- wx.LC_LIST
- wx.LC_REPORT
- wx.LC_VIRTUAL
- wx.LC_ICON
- wx.LC_SMALL_ICON
- wx.LC_ALIGN_LEFT
- wx.LC_EDIT_LABELS
- wx.LC_NO_HEADER
- wx.LC_SORT_ASCENDING

- wx.LC_SORT_DESCENDING
- wx.LC_HRULES
- wx.LC_VRULES

Simple example

In the first example we will introduce basic functionality of a *wx.ListCtrl*.

```
#!/usr/bin/python

# actresses.py

import wx
import sys

packages = [('jessica alba', 'pomona', '1981'), ('sigourney weaver', 'new york', '194
    ('angelina jolie', 'los angeles', '1975'), ('natalie portman', 'jerusalem', '1981
    ('rachel weiss', 'london', '1971'), ('scarlett johansson', 'new york', '1984' )]



class Actresses(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(380, 230))

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        panel = wx.Panel(self, -1)

        self.list = wx.ListCtrl(panel, -1, style=wx.LC_REPORT)
        self.list.InsertColumn(0, 'name', width=140)
        self.list.InsertColumn(1, 'place', width=130)
        self.list.InsertColumn(2, 'year', wx.LIST_FORMAT_RIGHT, 90)

        for i in packages:
            index = self.list.InsertStringItem(sys.maxint, i[0])
            self.list.SetStringItem(index, 1, i[1])
            self.list.SetStringItem(index, 2, i[2])

        hbox.Add(self.list, 1, wx.EXPAND)
        panel.SetSizer(hbox)

        self.Centre()
        self.Show(True)

app = wx.App()
Actresses(None, -1, 'actresses')
app.MainLoop()
```

```
self.list = wx.ListCtrl(panel, -1, style=wx.LC_REPORT)
```

We create a *wx.ListCtrl* with a wx.LC_REPORT style.

```
self.list.InsertColumn(0, 'name', width=140)
self.list.InsertColumn(1, 'place', width=130)
self.list.InsertColumn(2, 'year', wx.LIST_FORMAT_RIGHT, 90)
```

We insert three columns. We can specify the *width* of the column and the *format* of the column. The default format is *wx.LIST_FORMAT_LEFT*.

```
for i in packages:
    index = self.list.InsertStringItem(sys.maxint, i[0])
    self.list.SetStringItem(index, 1, i[1])
    self.list.SetStringItem(index, 2, i[2])
```

We insert data into the *wx.ListCtrl* using two methods. Each row begins with a *InsertStringItem()* method. The first parameter of the method specifies the row number. By giving a sys.maxint we ensure, that each call will insert data after the last row. The method returns the row index. The *SetStringItem()* method adds data to the consecutive columns of the current row.

Mixins

Mixins are classes that further enhance the functionality of a *wx.ListCtrl*. Mixin classes are so called helper classes. They are located in *wx.lib.mixins.listctrl* module. In order to use them, the programmer has to inherit from these classes.

There are five available mixins. As of 2.8.1.1.

- wx.ColumnSorterMixin
- wx.ListCtrlAutoWidthMixin
- wx.ListCtrlSelectionManagerMix
- wx.TextEditMixin
- wx.CheckListCtrlMixin

wx.ColumnSorterMixin is a mixin that enables sorting of columns in a report view. *wx.ListCtrlAutoWidthMixin* class automatically resizes the last column to the end of the *wx.ListCtrl*. By default, the last column does not take the remaining space. See the previous example. *wx.ListCtrlSelectionManagerMix* defines platform independent selection policy. *wx.TextEditMixin* enables text to be edited. *wx.CheckListCtrlMixin* adds a check box to each row. This way we can control rows. We can set every row to be checked or unchecked.

The following code shows, how we can use *ListCtrlAutoWidthMixin*

```
#!/usr/bin/python

# autowidth.py

import wx
import sys
from wx.lib.mixins.listctrl import ListCtrlAutoWidthMixin

actresses = [('jessica alba', 'pomona', '1981'), ('sigourney weaver', 'new york', '19
    ('angelina jolie', 'los angeles', '1975'), ('natalie portman', 'jerusalem', '1981
    ('rachel weiss', 'london', '1971'), ('scarlett johansson', 'new york', '1984' )]


class AutoWidthListCtrl(wx.ListCtrl, ListCtrlAutoWidthMixin):
    def __init__(self, parent):
        wx.ListCtrl.__init__(self, parent, -1, style=wx.LC_REPORT)
        ListCtrlAutoWidthMixin.__init__(self)
```

```
class Actresses(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(380, 230))

        hbox = wx.BoxSizer(wx.HORIZONTAL)

        panel = wx.Panel(self, -1)

        self.list = AutoWidthListCtrl(panel)
        self.list.InsertColumn(0, 'name', width=140)
        self.list.InsertColumn(1, 'place', width=130)
        self.list.InsertColumn(2, 'year', wx.LIST_FORMAT_RIGHT, 90)

        for i in actresses:
            index = self.list.InsertStringItem(sys.maxint, i[0])
            self.list.SetStringItem(index, 1, i[1])
            self.list.SetStringItem(index, 2, i[2])

        hbox.Add(self.list, 1, wx.EXPAND)
        panel.SetSizer(hbox)

        self.Centre()
        self.Show(True)

app = wx.App()
Actresses(None, -1, 'actresses')
app.MainLoop()
```

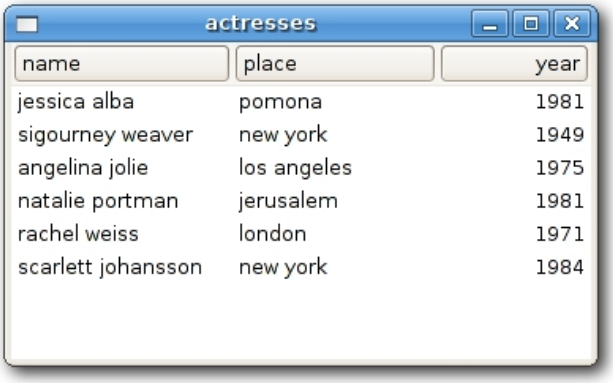
We change the previous example a bit.

```
from wx.lib.mixins.listctrl import ListCtrlAutoWidthMixin
```

Here we import the mixin.

```
class AutoWidthListCtrl(wx.ListCtrl, ListCtrlAutoWidthMixin):
    def __init__(self, parent):
        wx.ListCtrl.__init__(self, parent, -1, style=wx.LC_REPORT)
        ListCtrlAutoWidthMixin.__init__(self)
```

We create a new *AutoWidthListCtrl* class. This class will inherit from *wx.ListCtrl* and *ListCtrlAutoWidthMixin*. This is called **multiple inheritance**. The last column will automatically resize to take up the remaining width of a *wx.ListCtrl*.



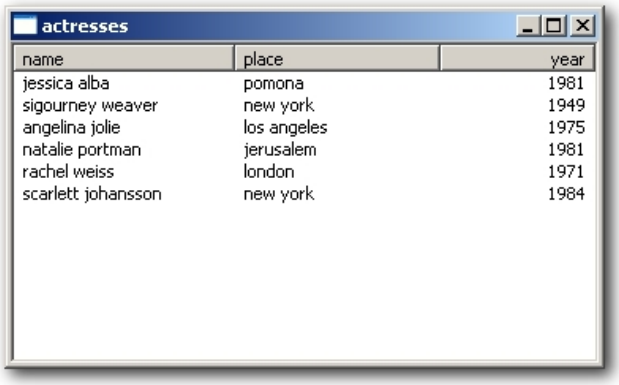


Figure: AutoWidth example

In the following example we will show, how we can create sortable columns. If we click on the column header, the corresponding rows in a column are sorted.

```
#!/usr/bin/python

# sorted.py

import wx
import sys
from wx.lib.mixins.listctrl import ColumnSorterMixin

actresses = {
1 : ('jessica alba', 'pomona', '1981'),
2 : ('sigourney weaver', 'new york', '1949'),
3 : ('angelina jolie', 'los angeles', '1975'),
4 : ('natalie portman', 'jerusalem', '1981'),
5 : ('rachel weiss', 'london', '1971'),
6 : ('scarlett johansson', 'new york', '1984')
}


class SortedListCtrl(wx.ListCtrl, ColumnSorterMixin):
    def __init__(self, parent):
        wx.ListCtrl.__init__(self, parent, -1, style=wx.LC_REPORT)
        ColumnSorterMixin.__init__(self, len(actresses))
        self.itemDataMap = actresses

    def GetListCtrl(self):
        return self

class Actresses(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(380, 230))

        hbox = wx.BoxSizer(wx.HORIZONTAL)

        panel = wx.Panel(self, -1)

        self.list = SortedListCtrl(panel)
        self.list.InsertColumn(0, 'name', width=140)
        self.list.InsertColumn(1, 'place', width=130)
        self.list.InsertColumn(2, 'year', wx.LIST_FORMAT_RIGHT, 90)

        items = actresses.items()

        for key, data in items:
```

```
            index = self.list.InsertStringItem(sys.maxint, data[0])
            self.list.SetStringItem(index, 1, data[1])
            self.list.SetStringItem(index, 2, data[2])
            self.list.SetItemData(index, key)

        hbox.Add(self.list, 1, wx.EXPAND)
        panel.SetSizer(hbox)

        self.Centre()
        self.Show(True)

app = wx.App()
Actresses(None, -1, 'actresses')
app.MainLoop()
```

We will again use the example with actresses.

```
ColumnSorterMixin.__init__(self, len(actresses))
```

The *ColumnSorterMixin* accepts one argument. It is the number of columns to be sorted.

```
self.itemDataMap = actresses
```

We must map our data to be displayed in a list control to the *itemDataMap* attribute. The data must be in a dictionary data type.

```
def GetListCtrl(self):
    return self
```

We must create a *GetListCtrl()* method. This method returns the *wx.ListCtrl* widget that is going to be sorted.

```
self.list.SetItemData(index, key)
```

We must assosiate each row with a special index. This is done with the *SetItemData* method.

Reader

A reader is a complex example showing two list controls in a report view.

```
#!/usr/bin/python

# reader.py


import wx

articles = [['Mozilla rocks', 'The year of the Mozilla', 'Earth on Fire'],
            ['Gnome pretty, Gnome Slow', 'Gnome, KDE, Icewm, XFCE', 'Where is Gnome h
            ['Java number one language', 'Compiled languages, intrepreted Languages',
```

```
class ListCtrlLeft(wx.ListCtrl):
    def __init__(self, parent, id):
        wx.ListCtrl.__init__(self, parent, id, style=wx.LC_REPORT | wx.LC_HRULES |
                wx.LC_NO_HEADER | wx.LC_SINGLE_SEL)
        images = ['icons/java.png', 'icons/gnome.png', 'icons/mozilla.png']

        self.parent = parent

        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_LIST_ITEM_SELECTED, self.OnSelect)

        self.il = wx.ImageList(32, 32)
        for i in images:
            self.il.Add(wx.Bitmap(i))

        self.SetImageList(self.il, wx.IMAGE_LIST_SMALL)
        self.InsertColumn(0, '')

        for i in range(3):
            self.InsertStringItem(0, '')
            self.SetItemImage(0, i)

    def OnSize(self, event):
        size = self.parent.GetSize()
        self.SetColumnWidth(0, size.x-5)
        event.Skip()

    def OnSelect(self, event):
        window = self.parent.GetGrandParent().FindWindowByName('ListControlOnRight')
        index = event.GetIndex()
        window.LoadData(index)

    def OnDeSelect(self, event):
        index = event.GetIndex()
        self.SetItemBackgroundColour(index, 'WHITE')

    def OnFocus(self, event):
        self.SetItemBackgroundColour(0, 'red')

class ListCtrlRight(wx.ListCtrl):
    def __init__(self, parent, id):
        wx.ListCtrl.__init__(self, parent, id, style=wx.LC_REPORT | wx.LC_HRULES |
                wx.LC_NO_HEADER | wx.LC_SINGLE_SEL)

        self.parent = parent

        self.Bind(wx.EVT_SIZE, self.OnSize)

        self.InsertColumn(0, '')


    def OnSize(self, event):
        size = self.parent.GetSize()
        self.SetColumnWidth(0, size.x-5)
        event.Skip()

    def LoadData(self, index):
        self.DeleteAllItems()
        for i in range(3):
            self.InsertStringItem(0, articles[index][i])


class Reader(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)
```

```
        hbox = wx.BoxSizer(wx.HORIZONTAL)
        splitter = wx.SplitterWindow(self, -1, style=wx.SP_LIVE_UPDATE|wx.SP_NOBORDER

        vbox1 = wx.BoxSizer(wx.VERTICAL)
        panel1 = wx.Panel(splitter, -1)
        panel11 = wx.Panel(panel1, -1, size=(-1, 40))
        panel11.SetBackgroundColour('#53728c')
        st1 = wx.StaticText(panel11, -1, 'Feeds', (5, 5))
        st1.SetForegroundColour('WHITE')

        panel12 = wx.Panel(panel1, -1, style=wx.BORDER_SUNKEN)
        vbox = wx.BoxSizer(wx.VERTICAL)
        list1 = ListCtrlLeft(panel12, -1)

        vbox.Add(list1, 1, wx.EXPAND)
        panel12.SetSizer(vbox)
        panel12.SetBackgroundColour('WHITE')


        vbox1.Add(panel11, 0, wx.EXPAND)
        vbox1.Add(panel12, 1, wx.EXPAND)

        panel1.SetSizer(vbox1)

        vbox2 = wx.BoxSizer(wx.VERTICAL)
        panel2 = wx.Panel(splitter, -1)
        panel21 = wx.Panel(panel2, -1, size=(-1, 40), style=wx.NO_BORDER)
        st2 = wx.StaticText(panel21, -1, 'Articles', (5, 5))
        st2.SetForegroundColour('WHITE')

        panel21.SetBackgroundColour('#53728c')
        panel22 = wx.Panel(panel2, -1, style=wx.BORDER_RAISED)
        vbox3 = wx.BoxSizer(wx.VERTICAL)
        list2 = ListCtrlRight(panel22, -1)
        list2.SetName('ListControlOnRight')
        vbox3.Add(list2, 1, wx.EXPAND)
        panel22.SetSizer(vbox3)


        panel22.SetBackgroundColour('WHITE')
        vbox2.Add(panel21, 0, wx.EXPAND)
        vbox2.Add(panel22, 1, wx.EXPAND)

        panel2.SetSizer(vbox2)

        toolbar = self.CreateToolBar()
        toolbar.AddLabelTool(1, 'Exit', wx.Bitmap('icons/stock_exit.png'))
        toolbar.Realize()

        self.Bind(wx.EVT_TOOL, self.ExitApp, id=1)

        hbox.Add(splitter, 1, wx.EXPAND | wx.TOP | wx.BOTTOM, 5)
        self.SetSizer(hbox)
        self.CreateStatusBar()
        splitter.SplitVertically(panel1, panel2)
        self.Centre()
        self.Show(True)


    def ExitApp(self, event):
        self.Close()


app = wx.App()
Reader(None, -1, 'Reader')
app.MainLoop()
```

The previous example showed a *wx.ListCtrl* in a report view. With no headers. We shall create our own headers. We show two *wx.ListCtrl* widgets. One is on the right side and the other one on the left side of the application.

```
splitter = wx.SplitterWindow(self, -1, style=wx.SP_LIVE_UPDATE|wx.SP_NOBORDER)
...
splitter.SplitVertically(panel1, panel2)
```

The splitter will split the main window into two vertical parts. The splitter will show two panels. Those two panels will have another two panels. They create *Feeds* and *Articles* headers. The rest of the space will be occupied by our two *wx.ListCtrl* widgets.

```
list2 = ListCtrlRight(panel22, -1)
list2.SetName('ListControlOnRight')
```

When we create *ListCtrlRight* object, we give it a name *ListControlOnRight*. This is because we need *ListCtrlRight* and *ListCtrlLeft* two widgets to communicate.

```
def OnSelect(self, event):
    window = self.parent.GetGrandParent().FindWindowByName('ListControlOnRight')
    index = event.GetIndex()
    window.LoadData(index)
```

This code is in *ListCtrlLeft* class. Here we locate the *ListCtrlRight* object and call it's *LoadData()* method.

```
def LoadData(self, index):
    self.DeleteAllItems()
    for i in range(3):
        self.InsertStringItem(0, articles[index][i])
```

The *LoadData()* method first clears all items. Then it inserts the article names from the globally defined articles list. The index has been passed.

```
def OnSize(self, event):
    size = self.parent.GetSize()
    self.SetColumnWidth(0, size.x-5)
    event.Skip()
```

Both *wx.ListCtrl*s have only one column. Here we ensure that the size of the column equals to size of the parent panel. The application would not look nice otherwise. Why do we extract 5px? This number is a kind of magic number. If we extract exactly 5px, the horizotal scrollbars do not appear. On other platforms, the number might be different.

Figure: Reader

CheckListCtrl

It is quite common to see applications having check boxes inside list controls. For example a packaging application like Synaptic or KYUM.

From the programmer's point of view, those checkboxes are simple images. There are two states. Checked and unchecked. For both situations we have a unique image. We do not have to implement the functionality. It has been already coded. The code is in *CheckListCtrlMixin*.

```
#!/usr/bin/python

# repository.py

import wx
import sys
from wx.lib.mixins.listctrl import CheckListCtrlMixin, ListCtrlAutoWidthMixin

packages = [('abiword', '5.8M', 'base'), ('adie', '145k', 'base'),
    ('airsnort', '71k', 'base'), ('ara', '717k', 'base'), ('arc', '139k', 'base'),
    ('asc', '5.8M', 'base'), ('ascii', '74k', 'base'), ('ash', '74k', 'base')]

class CheckListCtrl(wx.ListCtrl, CheckListCtrlMixin, ListCtrlAutoWidthMixin):
    def __init__(self, parent):
        wx.ListCtrl.__init__(self, parent, -1, style=wx.LC_REPORT | wx.SUNKEN_BORDER)
        CheckListCtrlMixin.__init__(self)
        ListCtrlAutoWidthMixin.__init__(self)


class Repository(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(450, 400))

        panel = wx.Panel(self, -1)

        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        leftPanel = wx.Panel(panel, -1)
        rightPanel = wx.Panel(panel, -1)

        self.log = wx.TextCtrl(rightPanel, -1, style=wx.TE_MULTILINE)
```

```
        self.list = CheckListCtrl(rightPanel)
        self.list.InsertColumn(0, 'Package', width=140)
        self.list.InsertColumn(1, 'Size')
        self.list.InsertColumn(2, 'Repository')

        for i in packages:
            index = self.list.InsertStringItem(sys.maxint, i[0])
            self.list.SetStringItem(index, 1, i[1])
            self.list.SetStringItem(index, 2, i[2])

        vbox2 = wx.BoxSizer(wx.VERTICAL)

        sel = wx.Button(leftPanel, -1, 'Select All', size=(100, -1))
        des = wx.Button(leftPanel, -1, 'Deselect All', size=(100, -1))
        apply = wx.Button(leftPanel, -1, 'Apply', size=(100, -1))


        self.Bind(wx.EVT_BUTTON, self.OnSelectAll, id=sel.GetId())
        self.Bind(wx.EVT_BUTTON, self.OnDeselectAll, id=des.GetId())
        self.Bind(wx.EVT_BUTTON, self.OnApply, id=apply.GetId())

        vbox2.Add(sel, 0, wx.TOP, 5)
        vbox2.Add(des)
        vbox2.Add(apply)

        leftPanel.SetSizer(vbox2)

        vbox.Add(self.list, 1, wx.EXPAND | wx.TOP, 3)
        vbox.Add((-1, 10))
        vbox.Add(self.log, 0.5, wx.EXPAND)
        vbox.Add((-1, 10))

        rightPanel.SetSizer(vbox)

        hbox.Add(leftPanel, 0, wx.EXPAND | wx.RIGHT, 5)
        hbox.Add(rightPanel, 1, wx.EXPAND)
        hbox.Add((3, -1))

        panel.SetSizer(hbox)

        self.Centre()
        self.Show(True)

    def OnSelectAll(self, event):
        num = self.list.GetItemCount()
        for i in range(num):
            self.list.CheckItem(i)

    def OnDeselectAll(self, event):
        num = self.list.GetItemCount()
        for i in range(num):
            self.list.CheckItem(i, False)

    def OnApply(self, event):
        num = self.list.GetItemCount()
        for i in range(num):
            if i == 0: self.log.Clear()
            if self.list.IsChecked(i):
                self.log.AppendText(self.list.GetItemText(i) + '\n')

app = wx.App()
Repository(None, -1, 'Repository')
app.MainLoop()
```

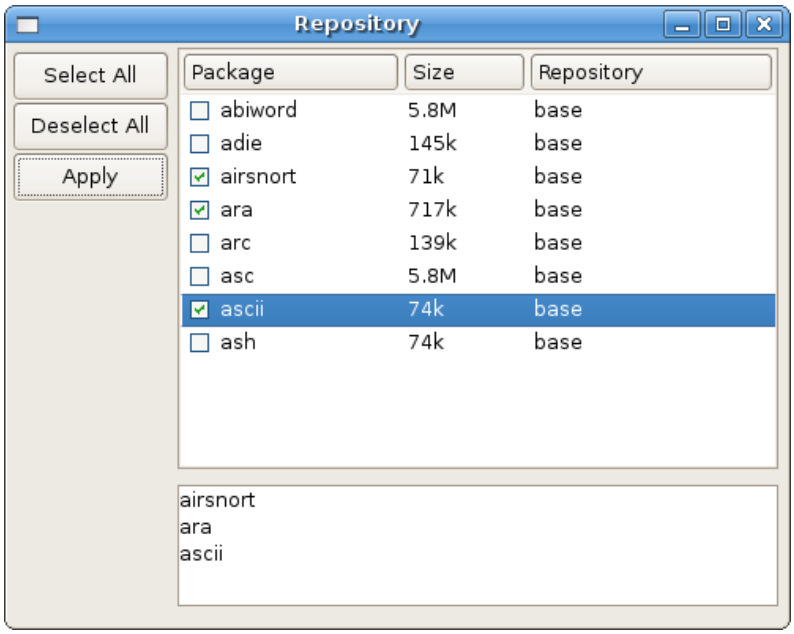


Figure: Repository

```
class CheckListCtrl(wx.ListCtrl, CheckListCtrlMixin, ListCtrlAutoWidthMixin):
    def __init__(self, parent):
        wx.ListCtrl.__init__(self, parent, -1, style=wx.LC_REPORT | wx.SUNKEN_BORDER)
        CheckListCtrlMixin.__init__(self)
        ListCtrlAutoWidthMixin.__init__(self)
```

wxPython enables multiple inheritance. Here we inherit from three different classes.

```
def OnSelectAll(self, event):
    num = self.list.GetItemCount()
    for i in range(num):
        self.list.CheckItem(i)
```

Here we can see multiple inheritance in action. We can call two methods from two different classes on our *self.list* object. The *GetItemCount()* method is located in *CheckListCtrl* class and the *CheckItem()* method is in *CheckListCtrlMixin* class.

Home ‡ Contents ‡ Top of Page

Drag and drop in wxPython

Wikipedia: In computer graphical user interfaces, drag-and-drop is the action of (or support for the action of) clicking on a virtual object and dragging it to a different location or onto another virtual object. In general, it can be used to invoke many kinds of actions, or create various types of associations between two abstract objects.

Drag and drop functionality is one of the most visible aspects of the graphical user interface. Drag and drop operation enables you to do complex things intuitively.

In drag and drop we basically drag some data from a data source to a data target. So we must have:

- Some data
- A data source
- A data target

In wxPython we have two predefined data targets. **wx.TextDropTarget** and **wx.FileDropTarget**.

- wx.TextDropTarget
- wx.FileDropTarget

wx.TextDropTarget

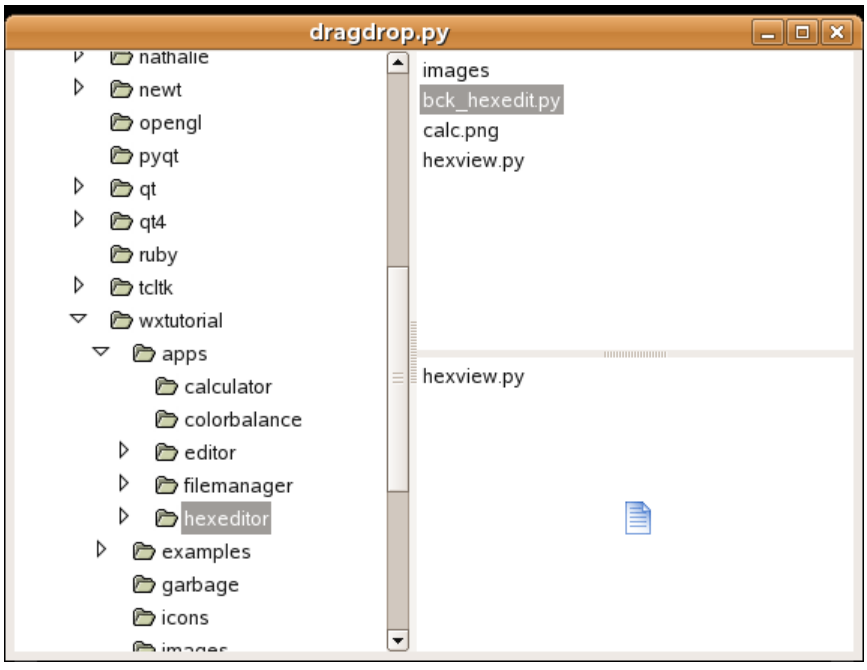


Figure: dragdrop.py

```
#!/usr/bin/python

# dragdrop.py

import os
import wx

class MyTextDropTarget(wx.TextDropTarget):
    def __init__(self, object):
        wx.TextDropTarget.__init__(self)
        self.object = object

    def OnDropText(self, x, y, data):
        self.object.InsertStringItem(0, data)


class DragDrop(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(650, 500))

        splitter1 = wx.SplitterWindow(self, -1, style=wx.SP_3D)
        splitter2 = wx.SplitterWindow(splitter1, -1, style=wx.SP_3D)
        self.dir = wx.GenericDirCtrl(splitter1, -1, dir='/home/', style=wx.DIRCTRL_DIR_ONL
        self.lc1 = wx.ListCtrl(splitter2, -1, style=wx.LC_LIST)
        self.lc2 = wx.ListCtrl(splitter2, -1, style=wx.LC_LIST)

        dt = MyTextDropTarget(self.lc2)
        self.lc2.SetDropTarget(dt)
        self.Bind(wx.EVT_LIST_BEGIN_DRAG, self.OnDragInit, id=self.lc1.GetId())

        tree = self.dir.GetTreeCtrl()

        splitter2.SplitHorizontally(self.lc1, self.lc2)
        splitter1.SplitVertically(self.dir, splitter2)

        self.Bind(wx.EVT_TREE_SEL_CHANGED, self.OnSelect, id=tree.GetId())

        self.OnSelect(0)
        self.Centre()
        self.Show(True)

    def OnSelect(self, event):
        list = os.listdir(self.dir.GetPath())
        self.lc1.ClearAll()
        self.lc2.ClearAll()
        for i in range(len(list)):
            if list[i][0] != '.':
                self.lc1.InsertStringItem(0, list[i])

    def OnDragInit(self, event):
        text = self.lc1.GetItemText(event.GetIndex())
        tdo = wx.TextDataObject(text)
        tds = wx.DropSource(self.lc1)
        tds.SetData(tdo)
        tds.DoDragDrop(True)


app = wx.App()
DragDrop(None, -1, 'dragdrop.py')
app.MainLoop()
```

wx.FileDropTarget

One of the advantages of the GUI over the console is it's intuitiveness. You can learn a GUI program easier than a console application. You often do not need a manual. On the other hand, some graphical operations are too complex. For

example, deleting a file by dragging it and droping it to the trash basket is very intuitive and easy to understand, but actually most people just press the delete key. (shift + delete) It is more effective. In our next example we explore a graphical operation, that is very handy. In most GUI text editors, you can open a file by simply dragging it from the file manager and dropping it on the editor.

```
#!/usr/bin/python

# filedrop.py

import wx

class FileDrop(wx.FileDropTarget):
    def __init__(self, window):
        wx.FileDropTarget.__init__(self)
        self.window = window

    def OnDropFiles(self, x, y, filenames):

        for name in filenames:
            try:
                file = open(name, 'r')
                text = file.read()
                self.window.WriteText(text)
                file.close()
            except IOError, error:
                dlg = wx.MessageDialog(None, 'Error opening file\n' + str(error))
                dlg.ShowModal()
            except UnicodeDecodeError, error:
                dlg = wx.MessageDialog(None, 'Cannot open non ascii files\n' + str(error))
                dlg.ShowModal()

class DropFile(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size = (450, 400))

        self.text = wx.TextCtrl(self, -1, style = wx.TE_MULTILINE)
        dt = FileDrop(self.text)
        self.text.SetDropTarget(dt)
        self.Centre()
        self.Show(True)


app = wx.App()
DropFile(None, -1, 'filedrop.py')
app.MainLoop()
```


Home ǂ Contents ǂ Top of Page

Internationalization

In computing, Internationalization and localization are means of adapting computer software for non-native environments, especially other nations and cultures. Internationalization is the process of ensuring that an application is capable of adapting to local requirements, for instance ensuring that the local writing system can be displayed. Localization is the process of adapting the software to be as familiar as possible to a specific locale, by displaying text in the local language and using local conventions for the display of such things as units of measurement. (wikipedia)

Unicode

There are two builds of wxPython. The ansi build and the unicode build. If we want to create and use wxPython applications in languages other than english, we must have the unicode build.

Unicode is an industry standard allowing computers to consistently represent and manipulate text expressed in any of the world's writing systems. It is a character enconding standart which uses 16 bits for storing characters. The traditional **ASCII** enconding uses only 8 bits.

First, we need to get the unicode enconding of Лев Николаевич Толстой Анна Каренина words.

```
>>> unicode(u'Лев Николаевич Толстой Анна Каренина')
u'\u041b\u0435\u0432 \u041d\u0438\u043a\u043e\u043b\u0430\u0435\u0432\u0438\u(
\u0422\u043e\u043b\u0441\u0442\u043e\u0439 \u0410\u043d\u043d\u0430
\u041a\u0430\u0440\u0435\u043d\u0438\u043d\u0430'
```

We launch the python terminal and use the *unicode()* function call. Notice, that in the example, we use additional \n\ characters to divide the words into two lines.

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

```
import wx

text = u'\u041b\u0435\u0432 \u041d\u0438\u043a\u043e\u043b\u0430\
\u0435\u0432\u0438\u0447 \u0422\u043e\u043b\u0441\u0442\u043e\u0439 \n\
\u0410\u043d\u043d\u0430 \u041a\u0430\u0440\u0435\u043d\u0438\u043d\u0430'


class Unicode(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        dc.DrawText(text, 50, 50)

app = wx.App()
Unicode(None, -1, 'Unicode')
app.MainLoop()
```

In the example, we draw Anna Karenina in russian azbuka on the window.

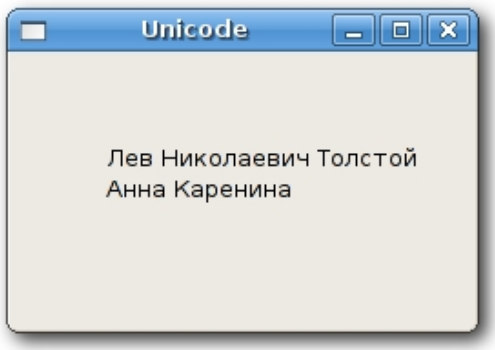


Figure: Unicode

Locale

A locale is an object that defines user's language, country, number format, letter format, currency format etc. A local variant has the following format.

```
[language[_territory][.codeset][@modifier]]
```

For example, **de_AT.utf8** is a german local used in Austria, with UTF8 codeset.

```
#!/usr/bin/python

# locale.py

import wx
import time
import locale

class Locale(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 420))

        panel = wx.Panel(self, -1)


        tm = time.localtime()

        font = wx.Font(10, wx.DEFAULT, wx.NORMAL, wx.BOLD)
        us = wx.StaticText(self, -1, 'United States', (25, 20))
        us.SetFont(font)

        wx.StaticLine(self, -1, (25, 50), (200 ,1))

        locale.setlocale(locale.LC_ALL, '')
        date = time.strftime('%x', tm)
        time_ = time.strftime('%X', tm)
        curr =  locale.currency(100000)

        wx.StaticText(self, -1, 'date: ', (25, 70))
        wx.StaticText(self, -1, 'time: ', (25, 90))
        wx.StaticText(self, -1, 'currency: ', (25, 110))

        wx.StaticText(self, -1, str(date), (125, 70))
        wx.StaticText(self, -1, str(time_), (125, 90))
        wx.StaticText(self, -1, str(curr), (125, 110))

        de = wx.StaticText(self, -1, 'Germany', (25, 150))
        de.SetFont(font)

        wx.StaticLine(self, -1, (25, 180), (200,1))

        locale.setlocale(locale.LC_ALL, ('de_DE', 'UTF8'))
        date = time.strftime('%x', tm)
        time_ = time.strftime('%X', tm)
        curr =  locale.currency(100000)

        wx.StaticText(self, -1, 'date: ', (25, 200))
        wx.StaticText(self, -1, 'time: ', (25, 220))
        wx.StaticText(self, -1, 'currency: ', (25, 240))
        wx.StaticText(self, -1, date, (125, 200))
        wx.StaticText(self, -1, time_, (125, 220))
        wx.StaticText(self, -1, curr, (125, 240))
```

```
        de = wx.StaticText(self, -1, 'Slovakia', (25, 280))
        de.SetFont(font)

        wx.StaticLine(self, -1, (25, 310), (200,1))

        locale.setlocale(locale.LC_ALL, ('sk_SK', 'UTF8'))
        date = time.strftime('%x', tm)
        time_ = time.strftime('%X', tm)
        curr =  locale.currency(100000)

        wx.StaticText(self, -1, 'date: ', (25, 330))
        wx.StaticText(self, -1, 'time: ', (25, 350))
        wx.StaticText(self, -1, 'currency: ', (25, 370))

        wx.StaticText(self, -1, str(date), (125, 330))
        wx.StaticText(self, -1, str(time_), (125, 350))
        wx.StaticText(self, -1, str(curr), (125, 370))

        self.Centre()
        self.Show(True)

app = wx.App()
Locale(None, -1, 'Locale')
app.MainLoop()
```

We use the standart built-in module **locale** to work with localized settings. In our example, we will show various formats of date, time and currency in the USA, Germany and Slovakia.

```
locale.setlocale(locale.LC_ALL, ('de_DE', 'UTF8'))
```

Here we set a locale object for Germany. **LC_ALL** is a combination of all various local settings, e.g. LC_TIME, LC_MONETARY, LC_NUMERIC.

```
date = time.strftime('%x', tm)
time_ = time.strftime('%X', tm)
curr =  locale.currency(100000)
```

These function calls reflect the current locale object.

Figure: Locale

World Time

At a specific moment, we have different time in countries across the world. Our globe is divided into time zones. It is not uncommon for programmers to deal with such tasks. wxPython comes with a *wx.DateTime* object. According to the documentation, wxDateTime class represents an absolute moment in the time.

```
#!/usr/bin/python

import wx
import time

class WorldTime(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(270, 280))

        self.panel = wx.Panel(self, -1)
        self.panel.SetBackgroundColour('#000000')
        font = wx.Font(12, wx.FONTFAMILY_DEFAULT,
                wx.FONTSTYLE_NORMAL, wx.FONTWEIGHT_BOLD, False, 'Georgia
```

```
        self.dt = wx.DateTime()

        self.tokyo = wx.StaticText(self.panel, -1,
                self.dt.FormatTime() , (20, 20))
        self.tokyo.SetForegroundColour('#23f002')
        self.tokyo.SetFont(font)

        self.moscow = wx.StaticText(self.panel, -1,
                self.dt.FormatTime() , (20, 70))
        self.moscow.SetForegroundColour('#23f002')
        self.moscow.SetFont(font)

        self.budapest = wx.StaticText(self.panel, -1,
                self.dt.FormatTime() , (20, 120))
        self.budapest.SetForegroundColour('#23f002')
        self.budapest.SetFont(font)

        self.london = wx.StaticText(self.panel, -1,
                self.dt.FormatTime() , (20, 170))
        self.london.SetForegroundColour('#23f002')
        self.london.SetFont(font)

        self.newyork = wx.StaticText(self.panel, -1,
                self.dt.FormatTime() , (20, 220))
        self.newyork.SetForegroundColour('#23f002')
        self.newyork.SetFont(font)

        self.OnTimer(None)

        self.timer = wx.Timer(self)
        self.timer.Start(1000)
        self.Bind(wx.EVT_TIMER, self.OnTimer)

        self.Centre()
        self.Show(True)

    def OnTimer(self, evt):
        now = self.dt.Now()
        self.tokyo.SetLabel('Tokyo: ' + str(now.Format(('%a %T'),
                wx.DateTime.GMT_9)))
        self.moscow.SetLabel('Moscow: ' + str(now.Format(('%a %T'),
                wx.DateTime.MSD)))
        self.budapest.SetLabel('Budapest: ' +  str(now.Format(('%a %T'),
                wx.DateTime.CEST)))
        self.london.SetLabel('London: ' + str(now.Format(('%a %T'),
                wx.DateTime.WEST)))
        self.newyork.SetLabel('New York: ' + str(now.Format(('%a %T'),
                wx.DateTime.EDT)))


app = wx.App()
WorldTime(None, -1, 'World Time')
app.MainLoop()
```

In the code example, we show current time in Tokyo, Moscow, Budapest, London and New York.

```
self.dt = wx.DateTime()
```

Here we create a *wx.DateTime* object.

```
now = self.dt.Now()
```

We get the "absolute moment" in time.

```
self.tokyo.SetLabel('Tokyo: ' + str(now.Format(('%a %T'),
    wx.DateTime.GMT_9)))
```

This code line sets the time to the appropriate format. The %a conversion specifier is an abbreviated weekday name according to the current locale. The %T is the time of day using decimal numbers using the format %H:%M:%S. The second parameter of the Format() method specifies the time zone. GMT_9 is used for Japan, EDT (Eastern Daylight Saving Time) is used in New York etc.

The code example was checked with the timeanddate.com website.

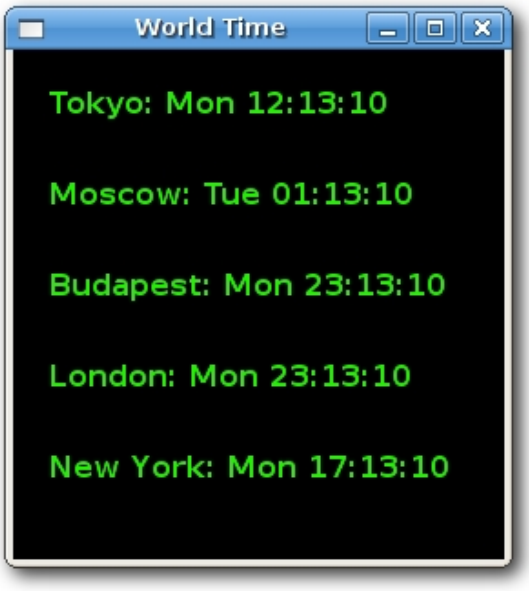


Figure: World Time

Sorting

Locale settings also affect the way, how strings are being sorted.

For example hungarian language has some characters that are missing in Slovak language or English language. Some languages have accents, others don't.

```
#!/usr/bin/python

# collate.py

import wx
import locale

ID_SORT = 1

words = [u'Sund', u'S\xe4bel', u'S\xfcnde', u'Schl\xe4fe', u'Sabotage']


class Collate(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 220))

        panel = wx.Panel(self, -1)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        self.listbox = wx.ListBox(panel, -1)
        for i in words:
            self.listbox.Append(i)
        hbox.Add(self.listbox, 1, wx.EXPAND | wx.ALL, 20)

        btnPanel = wx.Panel(panel, -1)
        vbox = wx.BoxSizer(wx.VERTICAL)
        new = wx.Button(btnPanel, ID_SORT, 'Sort', size=(90, 30))

        self.Bind(wx.EVT_BUTTON, self.OnSort, id=ID_SORT)

        vbox.Add((-1, 20))
        vbox.Add(new)

        btnPanel.SetSizer(vbox)
        hbox.Add(btnPanel, 0.6, wx.EXPAND | wx.RIGHT, 20)
        panel.SetSizer(hbox)

        locale.setlocale(locale.LC_COLLATE, ('de_DE', 'UTF8'))

        self.Centre()
        self.Show(True)

    def OnSort(self, event):
        self.listbox.Clear()
        words.sort( lambda a,b: locale.strcoll(a, b) )
        for i in words:
            self.listbox.Append(i)


app = wx.App()
Collate(None, -1, 'Collate')
app.MainLoop()
```

In our example, we took 5 germam words from the dictionary. The default *sort()* function sorts these words this way: Sabotage, Schläfe, Sund, Säbel, Sünde. This is incorrect, because in german alphabet ä character precedes a character. To get the corect sorting, we must use locale functions.

```
locale.setlocale(locale.LC_COLLATE, ('de_DE', 'UTF8'))
```

Here we set the german collate. We could use the *LC_ALL* option or the more specific *LC_COLLATE* one.

```
words.sort( lambda a,b: locale.strcoll(a, b) )
```

The trick is to use a new compare function within the *sort()* function. We define an anonymous lambda function. The *strcoll()* function compares two strings and returns -1, 0, 1 exactly like the default one, but it takes the locale settings (the collate) into account. This way we have the correct sorting of words.

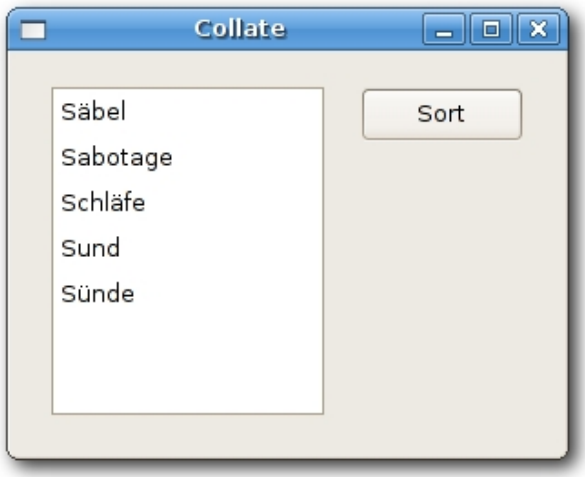


Figure: Collate

Simple Translation

In the following example, we will demonstrate a very basic translation.

A programmer has two options. Either to use the GNU gettext or to use the wxPython catalogs. Both systems are compatible.

wxPython has a class *wx.Locale*, which is a base for using

message catalogs. Each translation has one catalog. Say, we want to translate a string into german language. First, we must ensure, that we have language support for german language.

```
$ locale -a
C
de_AT.utf8
de_BE.utf8
de_CH.utf8
de_DE.utf8
de_LU.utf8
en_AU.utf8
en_BW.utf8
en_CA.utf8
en_DK.utf8
en_GB.utf8
en_HK.utf8
en_IE.utf8
en_IN
en_NZ.utf8
en_PH.utf8
en_SG.utf8
en_US.utf8
en_ZA.utf8
en_ZW.utf8
POSIX
sk_SK.utf8
```

To check what languages are supported, we use the *locale* command. On my system, I have english, german and slovak language support. English language and german language have different dialects, that's why we have so many options. Notice the **utf8** string. This means, that the system uses utf8 encoding for working with strings.

Next we write our code example. We put the string that are to be translated into this _(), or we can use the *wx.GetTranslation()* call.

```
#!/usr/bin/python

import wx

class Translation(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(220, 100))

        panel = wx.Panel(self, -1)

        mylocale = wx.Locale()
        mylocale.AddCatalogLookupPathPrefix('.')
```

```
        mylocale.AddCatalog('simple_de')

        _ = wx.GetTranslation

        wx.StaticText(panel, -1, _("hello"), (10, 10))
        #wx.StaticText(panel, -1, wx.GetTranslation('hello'), (10, 10))

        self.Centre()
        self.Show(True)

app = wx.App()
Translation(None, -1, 'Translation')
app.MainLoop()
```

Next we create a so called PO file. It is a simple text file, which is translators use to translate the strings.

```
pygettext -o simple_de.po simple.py
```

To create a po file, we use the **pygettext** command. To fully understand the format of the po file, consult the gnu gettext manual.

```
"Content-Type: text/plain; charset=utf-8\n"
```

We edit the simple_de.po file. We must specify the charset. In our case it is utf-8.

```
#: simple.py:17
msgid "hello"
msgstr "Grüß Gott"
```

Here we provide a translation for the hello string.

The last thing we do is to create a binary message catalog.

```
msgfmt --output-file simple_de.mo simple_de.po
```

To produce a mo file, we call the **msgfmt** command.

Figure: Simple translation

Home ‡ Contents ‡ Top of Page

Working with databases

Database driven applications account for a large part of all applications ever developed. And they will definitely in the future as well. Most of them are business applications. Companies work with large amount of data and they naturally need software for that. Well, you know, we live in a era called information revolution after all.

Some GUI toolkits are geared towards developing business applications. For example the WinForms or the Swing. They provide widgets that are adapted to the business application developing. A data grid widget is a good example. Swing toolkit has prorities like security or robustness. But definitelly not the look and feel.

Database is a structured collection of data that is stored in a computer. A computer program, that manages and queries a database is calles a **Database Management System (DBMS)**. Some thirty years ago, DBMS were available only in the research laboratories of giant companies like IBM. Later on, they began to spread. But they were very expensive. These days, we can found DBMS everywhere. On the web, on our personal computers, in various mobile devices or portables. We can have many different databeses for little or no money that would cost thousands of dollars in the past. We live in interesting times.

There are various database models. The most significant database model is the **relational database model (RDBMS)**. The data is divided into tables. Among these tables we define relations. We all have heard about various database management systems. There are several well known commercial DBMS as well as open source ones.

Commercial RDBMS	Opensource RDBMS
• Oracle • Sybase • MS SQL • Access	• MySQL • PostgreSQL • Firebird • SQLite

Python programming language has modules for all above RDBMS.

SQLite

Starting from Python 2.5.x series, an SQLite library is included in the python language. SQLite is a small embeddable library. This means that programmers can integrate the libraty inside their applications. No server is needed to work with SQLite. Therefore SQLite is also called a zero-configuration SQL database engine.

SQLite has the following features.

- works with transactions
- no administration needed
- small code footprint, less than 250 KB
- simple to use and fast
- single file database structure
- supports databases up to 2 tebibytes (2^{41} bytes) in size

SQLite supports these data types:

- TEXT
- INTEGER
- FLOAT
- BLOB
- NULL

Before we start working with SQLite, we define some important terms. A database **query** is a search for information from a database. A query is written in SQL language. **Structured Query Language** (SQL) is a computer language used to create, retrieve, update and delete data from the database. It was developed by the IBM corporation. SQL language has three subsets.

- DML
- DDL
- DCL

The DML (Data Manipulation Language) is used to add, update and delete data. SQLite understands insert, update and delete sql commands. The DDL (Data Definition Language) is used to define new tables and records. SQLite has create, drop, alter sql commands from this group. The DCL (Data Control Language) is used to set privileges for database users. SQLite does not have this subset.
A **cursor** is a database object used to traverse the results of a SQL query. A **transaction** is a unit of operation with a database management system. It can contain one or more queries. Transactions are used to ensure the integrity od data in a database. If everything is ok, transactions are commited. If one or more queries fails, transactions are rolled back. Databases that support

transactions are called transactional databases. An SQLite database is a transactional database. An SQL **result set** is a set of rows and metadata about the query from a database. It is a set of records that results from running a query. A single unit of structured data within a database table is called a **record** or a **row**.

sqlite3

SQLite library includes a small command line utility called sqlite3. It is used to manually enter and execute SQL commands against a SQLite database. To launch this utility, we type sqlite3 into the shell. The command is to be followed by a database name. If the database does not exist, a new one is created. We work with sqlite3 with a definite set of dot commands. To show all available commands, we type *.help*. Some of the commands are shown in the following table.

Command	Description
.databases	show a database name
.dump table	dump a table into an SQL text format
.exit	exit the sqlite3 program
.headers ON|OFF	show or hide column headers
.help	show help
.mode mode table	change mode for a table
.quit	same as .exit
.read filename	execute SQL commands in a filename
.show	show sqlite3 settings
.tables pattern	list tables that match pattern
.width num num ...	set width for columns

First, we create a new database called people.

```
$ sqlite3 people
SQLite version 3.3.13
Enter ".help" for instructions
sqlite>
sqlite> .databases
seq  name             file
---  ---------------  ----------------------------------------------------------
0    main             /home/vronskij/tmp/people
sqlite> .exit
$
```

All commands of *sqlite3* start with the dot "." character. To show all available commands, we simply type *.help*. The *.databases* command shows our current database. The *.exit* command quits the sqlite3 utility and returns to the shell.

Next we create a table.

```
sqlite> .tables
sqlite> create table neighbours(name text, age numeric, remark text);
sqlite> .tables
neighbours
```

The *.tables* command shows all available tables in the database. We create a table called neighbours. Our table will have three columns. We will use text and numeric data types. Notice that each SQL command is followed by a semicolon ";".

Now it is time to insert some real data.

```
sqlite> insert into neighbours values('sandy', 7, 'stubborn');
sqlite> insert into neighbours values('jane', 18, 'beautiful');
sqlite> insert into neighbours values('mark', 28, 'lazy');
sqlite> insert into neighbours values('steven', 34, 'friendly');
sqlite> insert into neighbours values('alice', 17, 'slick');
sqlite> insert into neighbours values('tom', 25, 'clever');
sqlite> insert into neighbours values('jack', 89, 'wise');
sqlite> insert into neighbours values('lucy', 18, 'cute');
```

The SQL *select* command is probably the most widely used DML (data manipulation language) command.

```
sqlite> select * from neighbours;
sandy|7|stubborn
jane|18|beautiful
mark|28|lazy
steven|34|friendly
alice|17|slick
tom|25|clever
jack|89|wise
lucy|18|cute
```

The sqlite3 has several modes to display data. Namely:

Mode	Description
csv	comma separated values
column	left aligned columns
html	html table code
insert	SQL insert statements for table
line	one value per line
list	values delimited by .separator string
tabs	tab separated values

The default mode is the list mode. We can see the current settings if we type the .show command.

```
sqlite> .show
     echo: off
  explain: off
  headers: off
     mode: list
nullvalue: ""
   output: stdout
separator: "|"
    width:
```

I prefer the column mode. In the next step we change the default settings a bit.

```
sqlite> .mode column
sqlite> .headers on
sqlite> .width 10 4 15
sqlite> select * from neighbours;
name        age   remark
----------  ----  ---------------
sandy       7     stubborn
jane        18    beautiful
mark        28    lazy
steven      34    friendly
alice       17    slick
tom         25    clever
jack        89    wise
lucy        18    cute
```

We change the mode with the *.mode* command to the column mode. We set headers on with the *.headers* command. Finally we change the width of each column with the *.width* command. The default value is ten characters.

Backing up the data is the most important thing. sqlite3 has a simple solution. We utilize command *.dump*.

```
sqlite> .tables
neighbours
sqlite> .dump neighbours
BEGIN TRANSACTION;
CREATE TABLE neighbours(name text, age numeric, remark text);
INSERT INTO "neighbours" VALUES('sandy',7,'stubborn');
INSERT INTO "neighbours" VALUES('jane',18,'beautiful');
INSERT INTO "neighbours" VALUES('mark',28,'lazy');
INSERT INTO "neighbours" VALUES('steven',34,'friendly');
INSERT INTO "neighbours" VALUES('alice',17,'slick');
INSERT INTO "neighbours" VALUES('tom',25,'clever');
INSERT INTO "neighbours" VALUES('jack',89,'wise');
INSERT INTO "neighbours" VALUES('lucy',18,'cute');
COMMIT;
```

The *.dump* command transforms the table into a set of SQL text

format. These SQL commands will recreate the table into the original state. We copy and paste these SQL commnads into a neighbours.sql text file.

In the next steps we drop a table and recreate it from our file.

```
sqlite> drop table neighbours;
sqlite> .tables
sqlite> .read ../neighbours.sql
sqlite> .tables
neighbours
sqlite> select * from neighbours;
name        age         remark
----------  ----------  ----------
sandy       7           stubborn
jane        18          beautiful
mark        28          lazy
steven      34          friendly
alice       17          slick
tom         25          clever
jack        89          wise
lucy        18          cute
```

We drop the neighbours table with the drop table SQL command. The command *.tables* shows no table. Then we type sqlite *.read* command to execute all SQL commands in the specified file. Finally, we verify our data.

SQLite python API

pysqlite is a python interface to the SQLite library. From python2.5x series, it is included in the python language. The pysqlite module is included under the package name sqlite3.

```
import sqlite3 as lite
```

Simple steps

- create connection object
- create cursor object
- execute query
- fetch data (optional)
- close cursor and connection objects

To create a connection, we call the *connect()* module method.

```
import sqlite3 as lite

con = lite.connect('databasename')
con = lite.connect(':memory:')
```

There are two ways for creating a connection object. We can create a connection to a database on the filesystem. We simply specify the path to the filename. We can also create a database in memory. This is done with a special string ':memory:'.

We launch a python interpreter. We will test our examples there.

```
$ python
Python 2.5.1c1 (release25-maint, Apr  6 2007, 22:02:36)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

```
>>> import sqlite3 as lite
>>> con = lite.connect('people')
>>> cur = con.cursor()
>>> cur.execute('select name from neighbours')
>>> print cur.fetchall()
[(u'sandy',), (u'jane',), (u'mark',), (u'steven',), (u'alice',), (u'tom',),
>>> cur.close()
>>> con.close()
```

First we import the sqlite3 module. Then we connect to our people database. The database file is in our current directory. To create a cursor object, we call the *cursor()* method of the connection object. After that we call two cursor object methods. The *execute()* method executes SQL commands. The *fetchall()* method retrieves all data that we have selected. The kosher way to end our work is to close the cursor and the connection object.

Commiting changes

SQLite library works with transactions. It is important to understand how it works. According to the documentation, for every DML statement, SQLite opens up a transaction. We must commit our changes to apply them. For every DCL statement, SQLite library commits automatically the changes. We will demonstrate this in short examples.

```
>>> cur.execute("update neighbours set age=29 where name='lucy'")
>>> cur.execute("select age from neighbours where name='lucy'")
>>> print cur.fetchone()
(29,)
>>> cur.close()
>>> con.close()
>>> (CTRL + D)
$ sqlite3 people
sqlite> select age from neighbours where name='lucy';
18
```

What went wrong? We did not commit our changes. When we executed the select statement using the python sqlite api, we received result within a transaction context. The changes were not really written to the database. When we checked the data in the sqlite3 utility, we got age 18. The data was not changed.

```
>>> cur.execute("update neighbours set age=29 where name='lucy'")
>>> con.commit()
>>> cur.close()
>>> con.close()
>>> (CTRL + D)
$ sqlite3 people
sqlite> select age from neighbours where name='lucy';
29
```

After committing our changes with the *commit()* method of the connection object, the data changes are really written to the database.

In the next example we demonstrate that the DCL statements are committed automatically. We will use create table command, which is a part of the DCL language.

```
>>> cur.execute('create table relatives(name text, age numeric)')
>>> cur.close()
>>> con.close()
>>> (CTRL + D)
$ sqlite3 people
sqlite> .tables
neighbours  relatives
```

There is one more thing to mention. We can create a connection, which will automatically commit all our changes. This is done, when we set the *isolation_level* parameter to None.

```
>>> import sqlite3 as lite
>>> con = lite.connect('people', isolation_level=None)
>>> cur = con.cursor()
>>> cur.execute("insert into neighbours values ('rebecca', 16, 'shy')")
>>> cur.close()
>>> con.close()
>>> (CTRL + D)
$ sqlite3 people
sqlite> select * from neighbours where name='rebecca';
rebecca|16|shy
sqlite>
```

Autoincrement

Autoincremental primary key is a handy feature. We insert new rows

and the key is incremented automatically by one. The implementation of the autoincrement feature may differ among RDMSs. In the next example we will show how it is done in SQLite database.

```
sqlite> create table books(id integer primary key autoincrement not null, na
sqlite> insert into books (name, author) values ('anna karenina', 'leo tolst
sqlite> insert into books (name, author) values ('father goriot', 'honore de
sqlite> select * from books;
1|anna karenina|leo tolstoy
2|father goriot|honore de balzac
sqlite>
```

The keyword *autoincrement* is used to create autoincremental primary key in SQLite.

Security considerations

It is possible but insecure to pass parameters this way.

```
bookname = 'atlante illustrato di filosofia'
bookauthor = 'ubaldo nicola'
cur.execute("insert into books(name, author) values ('%s', '%s')" % (booknam
```

It is vulnerable to attacks. These attacks are called SQL injections. Don't do this.

```
>>> import sqlite3 as lite
>>> print lite.paramstyle
qmark
```

The python Database API specification lists these possible parameter style passings:

- qmark
- numeric
- named
- format
- pyformat

Python SQLite API uses the qmark (question mark) quoting. The previous example rewritten in qmark style:

```
bookname = 'atlante illustrato di filosofia'
bookauthor = 'ubaldo nicola'
cur.execute('insert into books(name, author) values (?, ?)', (bookname, boo
```

TODO blob

Putting it together

So far we have been looking at the SQLite3 library, databases and SQL language. Now it is time to put it all together with wxPython in a simple functional script. The next simple script will do only one specific thing. Insert data into a table. We will use our people database, neigbours table.

```
#!/usr/bin/python

# insertdata.py

import wx
import sqlite3 as lite

class InsertData(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(280, 200))

        panel = wx.Panel(self, -1)

        gs = wx.FlexGridSizer(3, 2, 9, 9)
        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox = wx.BoxSizer(wx.HORIZONTAL)

        name = wx.StaticText(panel, -1, 'Name')
        remark = wx.StaticText(panel, -1, 'Remark')
        age = wx.StaticText(panel, -1, 'Age')
        self.sp = wx.SpinCtrl(panel, -1, '', size=(60, -1), min=1, max=125)
        self.tc1 = wx.TextCtrl(panel, -1, size=(150, -1))
        self.tc2 = wx.TextCtrl(panel, -1, size=(150, -1))

        gs.AddMany([(name), (self.tc1, 1, wx.LEFT, 10),
                (remark), (self.tc2, 1, wx.LEFT, 10),
                (age), (self.sp, 0, wx.LEFT, 10)])

        vbox.Add(gs, 0, wx.ALL, 10)
        vbox.Add((-1, 30))

        insert = wx.Button(panel, -1, 'Insert', size=(-1, 30))
        cancel = wx.Button(panel, -1, 'Cancel', size=(-1, 30))
        hbox.Add(insert)
        hbox.Add(cancel, 0, wx.LEFT, 5)
        vbox.Add(hbox, 0, wx.ALIGN_CENTER | wx.BOTTOM, 10)

        self.Bind(wx.EVT_BUTTON, self.OnInsert, id=insert.GetId())
        self.Bind(wx.EVT_BUTTON, self.OnCancel, id=cancel.GetId())

        panel.SetSizer(vbox)

        self.Centre()
        self.Show(True)

    def OnInsert(self, event):
        try:
            con = lite.connect('people')
            cur = con.cursor()
```

```
            name = self.tc1.GetValue()
            age = self.sp.GetValue()
            remark = self.tc2.GetValue()
            cur.execute('insert into neighbours values(?, ?, ?)', (name, age
            con.commit()
            cur.close()
            con.close()

        except lite.Error, error:
            dlg = wx.MessageDialog(self, str(error), 'Error occured')
            dlg.ShowModal()

    def OnCancel(self, event):
        self.Close()

app = wx.App()
InsertData(None, -1, 'Insert Dialog')
app.MainLoop()
```

```
gs = wx.FlexGridSizer(3, 2, 9, 9)
```

In our Dialog box we use items of different size. That's why we have chosen the *wx.FlexGridSizer.* Items in *wx.GridSizer* have always the same size.

```
name = self.tc1.GetValue()
age = self.sp.GetValue()
remark = self.tc2.GetValue()
cur.execute('insert into neighbours values(?, ?, ?)', (name, age, remark))
```

This is the crutial part of the code. In the first three lines, we get the values that the user has inserted. These values are inserted into the database with the appropriate the SQL code.

```
except lite.Error, error:
    dlg = wx.MessageDialog(self, str(error), 'Error occured')
    dlg.ShowModal()
```

We have placed our database related code between the try - catch clause. This is because working with data and databases is prone to errors. The *Error* exception is a base class for all other exceptions implemented in SQLite library.

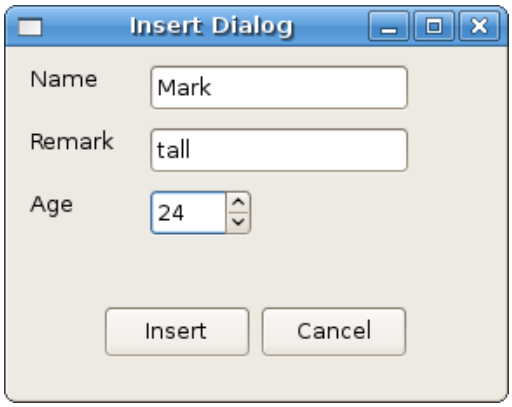


Figure: insertdata.py dialog

TODO: Supported Errors, Warning

Home ǂ Contents ǂ Top of Page

Application skeletons in wxPython

In this section, we will create some application skeletons. Our scripts will work out the interface but will not implement the functionality. The goal is to show, how several well known GUI interfaces could be done in wxPython. Most manuals, tutorials and books show only the basic usage of a widget. When I was a beginner, I always wondered how this or this could be done. And I think, many newbies think the same.

File Manager

File Hunter is a skeleton of a file manager. It copies the lookout of the Krusader, the best file manager available on Unix systems. If you double click on the splitter widget, it will divide the File Hunter into two parts with the same width. The same happens, if you resize the main window.

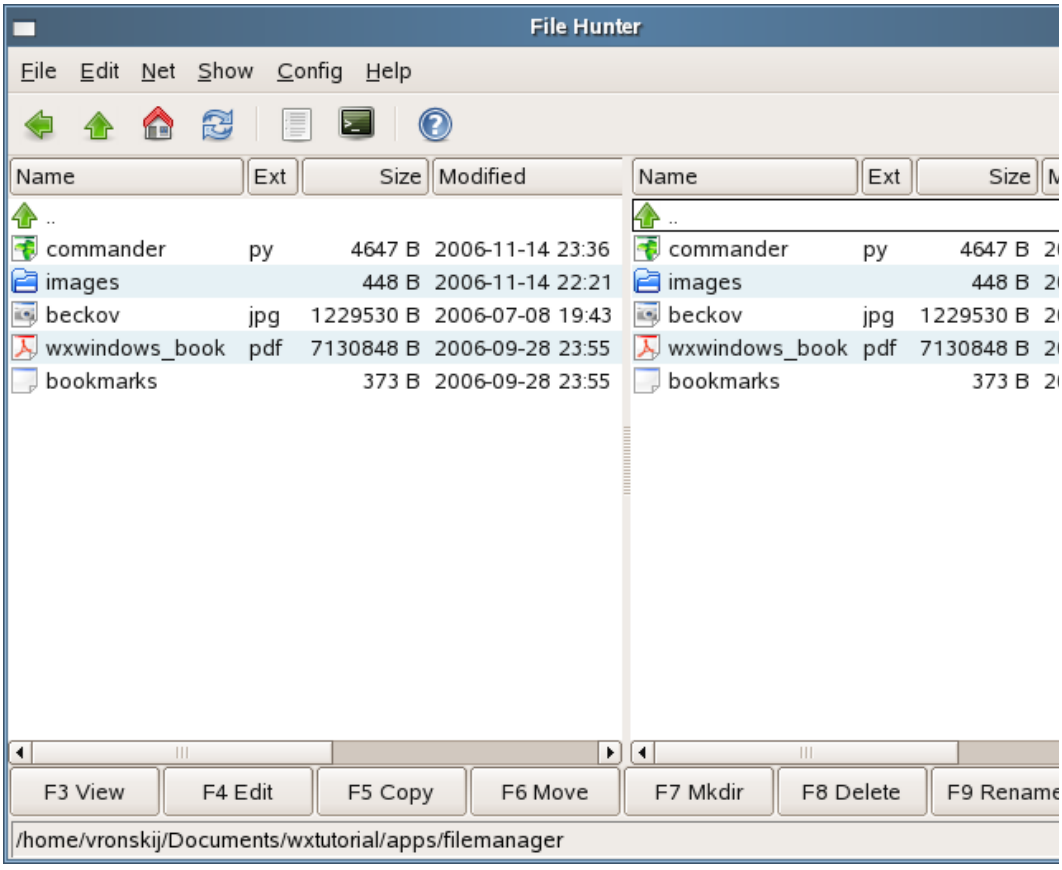


Figure: Filemanager.py

```
#!/usr/bin/python

import wx
import os
import time


ID_BUTTON=100
ID_EXIT=200
ID_SPLITTER=300

class MyListCtrl(wx.ListCtrl):
    def __init__(self, parent, id):
        wx.ListCtrl.__init__(self, parent, id, style=wx.LC_REPORT)

        files = os.listdir('.')
        images = ['images/empty.png', 'images/folder.png', 'images/source_py.png
                'images/image.png', 'images/pdf.png', 'images/up16.png']

        self.InsertColumn(0, 'Name')
        self.InsertColumn(1, 'Ext')
        self.InsertColumn(2, 'Size', wx.LIST_FORMAT_RIGHT)
        self.InsertColumn(3, 'Modified')

        self.SetColumnWidth(0, 220)
        self.SetColumnWidth(1, 70)
        self.SetColumnWidth(2, 100)
        self.SetColumnWidth(3, 420)

        self.il = wx.ImageList(16, 16)
        for i in images:
            self.il.Add(wx.Bitmap(i))
        self.SetImageList(self.il, wx.IMAGE_LIST_SMALL)

        j = 1
        self.InsertStringItem(0, '..')
        self.SetItemImage(0, 5)

        for i in files:
            (name, ext) = os.path.splitext(i)
            ex = ext[1:]
            size = os.path.getsize(i)
            sec = os.path.getmtime(i)
            self.InsertStringItem(j, name)
            self.SetStringItem(j, 1, ex)
            self.SetStringItem(j, 2, str(size) + ' B')
            self.SetStringItem(j, 3, time.strftime('%Y-%m-%d %H:%M',
                time.localtime(sec)))

            if os.path.isdir(i):
                self.SetItemImage(j, 1)
            elif ex == 'py':
                self.SetItemImage(j, 2)
            elif ex == 'jpg':
                self.SetItemImage(j, 3)
            elif ex == 'pdf':
                self.SetItemImage(j, 4)
            else:
                self.SetItemImage(j, 0)

            if (j % 2) == 0:
                self.SetItemBackgroundColour(j, '#e6f1f5')
```

```
            j = j + 1


class FileHunter(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, -1, title)

        self.splitter = wx.SplitterWindow(self, ID_SPLITTER, style=wx.SP_BORDER)
        self.splitter.SetMinimumPaneSize(50)

        p1 = MyListCtrl(self.splitter, -1)
        p2 = MyListCtrl(self.splitter, -1)
        self.splitter.SplitVertically(p1, p2)

        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_SPLITTER_DCLICK, self.OnDoubleClick, id=ID_SPLITTER)

        filemenu= wx.Menu()
        filemenu.Append(ID_EXIT,"E&xit"," Terminate the program")
        editmenu = wx.Menu()
        netmenu = wx.Menu()
        showmenu = wx.Menu()
        configmenu = wx.Menu()
        helpmenu = wx.Menu()

        menuBar = wx.MenuBar()
        menuBar.Append(filemenu,"&File")
        menuBar.Append(editmenu, "&Edit")
        menuBar.Append(netmenu, "&Net")
        menuBar.Append(showmenu, "&Show")
        menuBar.Append(configmenu, "&Config")
        menuBar.Append(helpmenu, "&Help")
        self.SetMenuBar(menuBar)
        self.Bind(wx.EVT_MENU, self.OnExit, id=ID_EXIT)

        tb = self.CreateToolBar( wx.TB_HORIZONTAL | wx.NO_BORDER |
                wx.TB_FLAT | wx.TB_TEXT)
        tb.AddSimpleTool(10, wx.Bitmap('images/previous.png'), 'Previous')
        tb.AddSimpleTool(20, wx.Bitmap('images/up.png'), 'Up one directory')
        tb.AddSimpleTool(30, wx.Bitmap('images/home.png'), 'Home')
        tb.AddSimpleTool(40, wx.Bitmap('images/refresh.png'), 'Refresh')
        tb.AddSeparator()
        tb.AddSimpleTool(50, wx.Bitmap('images/write.png'), 'Editor')
        tb.AddSimpleTool(60, wx.Bitmap('images/terminal.png'), 'Terminal')
        tb.AddSeparator()
        tb.AddSimpleTool(70, wx.Bitmap('images/help.png'), 'Help')
        tb.Realize()

        self.sizer2 = wx.BoxSizer(wx.HORIZONTAL)

        button1 = wx.Button(self, ID_BUTTON + 1, "F3 View")
        button2 = wx.Button(self, ID_BUTTON + 2, "F4 Edit")
        button3 = wx.Button(self, ID_BUTTON + 3, "F5 Copy")
        button4 = wx.Button(self, ID_BUTTON + 4, "F6 Move")
        button5 = wx.Button(self, ID_BUTTON + 5, "F7 Mkdir")
        button6 = wx.Button(self, ID_BUTTON + 6, "F8 Delete")
        button7 = wx.Button(self, ID_BUTTON + 7, "F9 Rename")
        button8 = wx.Button(self, ID_EXIT, "F10 Quit")

        self.sizer2.Add(button1, 1, wx.EXPAND)
        self.sizer2.Add(button2, 1, wx.EXPAND)
        self.sizer2.Add(button3, 1, wx.EXPAND)
        self.sizer2.Add(button4, 1, wx.EXPAND)
        self.sizer2.Add(button5, 1, wx.EXPAND)
```

```
        self.sizer2.Add(button6, 1, wx.EXPAND)
        self.sizer2.Add(button7, 1, wx.EXPAND)
        self.sizer2.Add(button8, 1, wx.EXPAND)

        self.Bind(wx.EVT_BUTTON, self.OnExit, id=ID_EXIT)

        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.splitter,1,wx.EXPAND)
        self.sizer.Add(self.sizer2,0,wx.EXPAND)
        self.SetSizer(self.sizer)

        size = wx.DisplaySize()
        self.SetSize(size)

        self.sb = self.CreateStatusBar()
        self.sb.SetStatusText(os.getcwd())
        self.Center()
        self.Show(True)


    def OnExit(self,e):
        self.Close(True)

    def OnSize(self, event):
        size = self.GetSize()
        self.splitter.SetSashPosition(size.x / 2)
        self.sb.SetStatusText(os.getcwd())
        event.Skip()


    def OnDoubleClick(self, event):
        size =  self.GetSize()
        self.splitter.SetSashPosition(size.x / 2)

app = wx.App(0)
FileHunter(None, -1, 'File Hunter')
app.MainLoop()
```

SpreadSheet

Gnumeric, KSpread and OpenOffice Calc are famous spreadsheet applications available on Unix. The following example shows a skeleton of a spreadsheet application in wxPython.

Applications have their own life. This is also true for educational scripts. After uprading to wx.Python 2.8.1.1 I realized, that the spreadsheet example does not work. The following line was the problem.

```
toolbar2.AddControl(wx.StaticText(toolbar2, -1, '  '))
```

Of course, we cannot add a widget to itself. But the previous version of the toolkit was happy with it. Under the current version it did not work, signalizing a problem. It might or might not work on the Mac and Windows. Originally, I wanted to add some space between the combo boxes. Under the new version of the toolkit it stopped to work either so I dropped the line.

Besides fixing this bug, I also cleaned the code a bit and replaced the depreciated methods (*AddSimpleTool()*) of the toolbar with the new ones (*AddLabelTool()*).

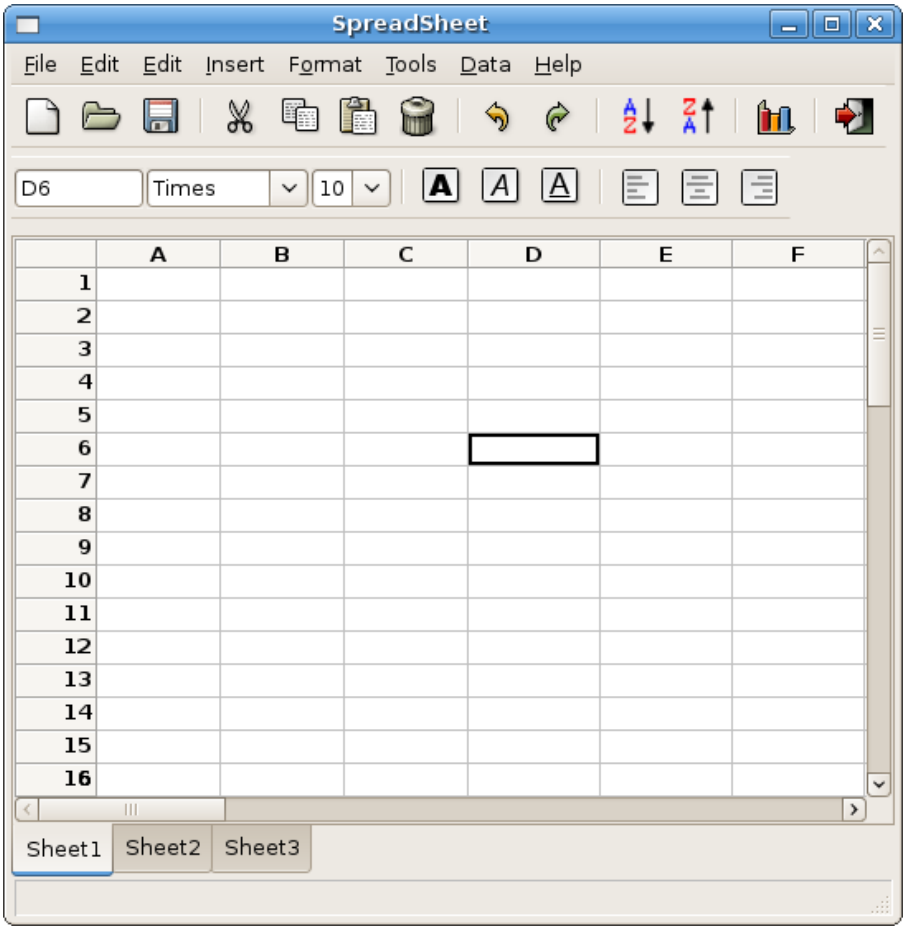


Figure: Spreadsheet

```
#!/usr/bin/python

# spreadsheet.py

from wx.lib import sheet
import wx


class MySheet(sheet.CSheet):
    def __init__(self, parent):
        sheet.CSheet.__init__(self, parent)
        self.row = self.col = 0
        self.SetNumberRows(55)
        self.SetNumberCols(25)

        for i in range(55):
            self.SetRowSize(i, 20)

    def OnGridSelectCell(self, event):
        self.row, self.col = event.GetRow(), event.GetCol()
```

```
        control = self.GetParent().GetParent().position
        value =  self.GetColLabelValue(self.col) + self.GetRowLabelValue(self.ro
        control.SetValue(value)
        event.Skip()

class Newt(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, -1, title, size = (550, 500))

        fonts = ['Times New Roman', 'Times', 'Courier', 'Courier New', 'Helvetic
                'Sans', 'verdana', 'utkal', 'aakar', 'Arial']
        font_sizes = ['10', '11', '12', '14', '16']

        box = wx.BoxSizer(wx.VERTICAL)
        menuBar = wx.MenuBar()

        menu1 = wx.Menu()
        menuBar.Append(menu1, '&File')
        menu2 = wx.Menu()
        menuBar.Append(menu2, '&Edit')
        menu3 = wx.Menu()
        menuBar.Append(menu3, '&Edit')
        menu4 = wx.Menu()
        menuBar.Append(menu4, '&Insert')
        menu5 = wx.Menu()
        menuBar.Append(menu5, 'F&ormat')
        menu6 = wx.Menu()
        menuBar.Append(menu6, '&Tools')
        menu7 = wx.Menu()
        menuBar.Append(menu7, '&Data')
        menu8 = wx.Menu()
        menuBar.Append(menu8, '&Help')

        self.SetMenuBar(menuBar)

        toolbar1 = wx.ToolBar(self, -1, style= wx.TB_HORIZONTAL)
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_new.png'))
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_open.png'))
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_save.png'))
        toolbar1.AddSeparator()
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_cut.png'))
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_copy.png'))
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_paste.png'))
        toolbar1.AddLabelTool(-1, '',  wx.Bitmap('icons/stock_delete.png'))
        toolbar1.AddSeparator()
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_undo.png'))
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/stock_redo.png'))
        toolbar1.AddSeparator()
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/incr22.png'))
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/decr22.png'))
        toolbar1.AddSeparator()
        toolbar1.AddLabelTool(-1, '', wx.Bitmap('icons/chart.xpm'))
        toolbar1.AddSeparator()
        toolbar1.AddLabelTool(-1, '',  wx.Bitmap('icons/stock_exit.png'))

        toolbar1.Realize()

        toolbar2 = wx.ToolBar(self, wx.TB_HORIZONTAL | wx.TB_TEXT)

        self.position = wx.TextCtrl(toolbar2)
        font = wx.ComboBox(toolbar2, -1, value = 'Times', choices=fonts, size=(1
                style=wx.CB_DROPDOWN)
        font_height = wx.ComboBox(toolbar2, -1, value = '10',  choices=font_size
                size=(50, -1), style=wx.CB_DROPDOWN)
```

```
        toolbar2.AddControl(self.position)
        toolbar2.AddControl(font)
        toolbar2.AddControl(font_height)
        toolbar2.AddSeparator()
        bold = wx.Bitmap('icons/stock_text_bold.png')
        toolbar2.AddCheckTool(-1, bold)
        italic = wx.Bitmap('icons/stock_text_italic.png')
        toolbar2.AddCheckTool(-1, italic)
        under = wx.Bitmap('icons/stock_text_underline.png')
        toolbar2.AddCheckTool(-1, under)
        toolbar2.AddSeparator()
        toolbar2.AddLabelTool(-1, '', wx.Bitmap('icons/text_align_left.png'))
        toolbar2.AddLabelTool(-1, '', wx.Bitmap('icons/text_align_center.png'))
        toolbar2.AddLabelTool(-1, '', wx.Bitmap('icons/text_align_right.png'))

        box.Add(toolbar1, border=5)
        box.Add((5,5) , 0)
        box.Add(toolbar2)
        box.Add((5,10) , 0)

        toolbar2.Realize()
        self.SetSizer(box)
        notebook = wx.Notebook(self, -1, style=wx.RIGHT)

        sheet1 = MySheet(notebook)
        sheet2 = MySheet(notebook)
        sheet3 = MySheet(notebook)
        sheet1.SetFocus()

        notebook.AddPage(sheet1, 'Sheet1')
        notebook.AddPage(sheet2, 'Sheet2')
        notebook.AddPage(sheet3, 'Sheet3')

        box.Add(notebook, 1, wx.EXPAND)

        self.CreateStatusBar()
        self.Centre()
        self.Show(True)

app = wx.App()
Newt(None, -1, 'SpreadSheet')
app.MainLoop()
```

Much of the code builds the menus and toolbars. Besides, it is quite a simple example.

```
class MySheet(sheet.CSheet):
    def __init__(self, parent):
        sheet.CSheet.__init__(self, parent)
        self.row = self.col = 0
        self.SetNumberRows(55)
        self.SetNumberCols(25)

        for i in range(55):
            self.SetRowSize(i, 20)
```

The MySheet class inherits from the CSheet class, which is located in

thel wx.lib module. It is basically a wx.Grid widget with some additional functionality. We set the row size to 20px. This is purely for aesthetical purpose.

```
control = self.GetParent().GetParent().position
```

The position text control shows the selected cell of the grid widget. It is the first widget of the second toolbar. Being inside a MySheet class, we need to get a reference to the text control, which is defined in the Newt class. MySheet is a child of the notebook. And notebook is a child of Newt. So we manage to get to the position text control by calling the *GetParent()* method twice.

```
notebook = wx.Notebook(self, -1, style=wx.RIGHT)
```

This is a bug. Under current version of wxPython (on GTK+), right is bottom and bottom is right.

Browser

These days internet browsers are one of the most important applications in the IT world. The best available browsers are Opera and Firefox. We mimic the look of a Firefox in our script.

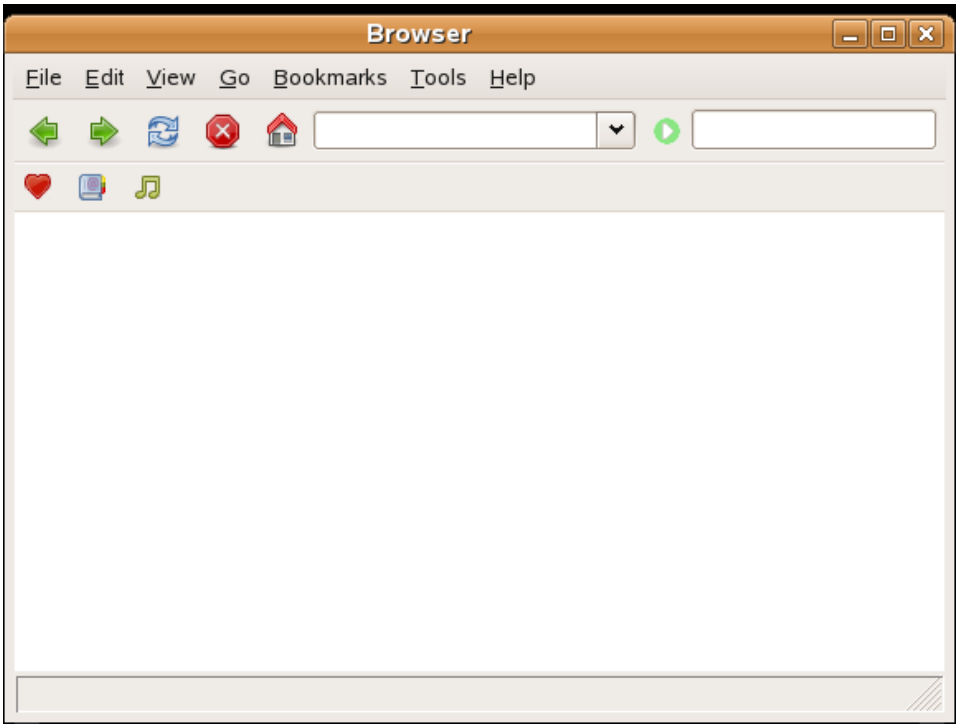


Figure: Browser.py

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

```
import wx
from wx.lib.buttons import GenBitmapTextButton

class Browser(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(450, 400))
        panel = wx.Panel(self, -1)
        panel.SetBackgroundColour('WHITE')

        menubar = wx.MenuBar()
        file = wx.Menu()
        file.Append(1, '&Quit', '')
        edit = wx.Menu()
        view = wx.Menu()
        go = wx.Menu()
        bookmarks = wx.Menu()
        tools = wx.Menu()
        help = wx.Menu()

        menubar.Append(file, '&File')
        menubar.Append(edit, '&Edit')
        menubar.Append(view, '&View')
        menubar.Append(go, '&Go')
        menubar.Append(bookmarks, '&Bookmarks')
        menubar.Append(tools, '&Tools')
        menubar.Append(help, '&Help')

        self.SetMenuBar(menubar)

        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox1 = wx.BoxSizer(wx.HORIZONTAL)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)
        toolbar1 = wx.Panel(panel, -1, size=(-1, 40))
        back = wx.BitmapButton(toolbar1, -1, wx.Bitmap('icons/back.png'),
                style=wx.NO_BORDER)
        forward = wx.BitmapButton(toolbar1, -1, wx.Bitmap('icons/forward.png'),
                style=wx.NO_BORDER)
        refresh = wx.BitmapButton(toolbar1, -1, wx.Bitmap('icons/refresh.png'),
                style=wx.NO_BORDER)
        stop = wx.BitmapButton(toolbar1, -1, wx.Bitmap('icons/stop.png'),
                style=wx.NO_BORDER)
        home = wx.BitmapButton(toolbar1, -1, wx.Bitmap('icons/home.png'),
                style=wx.NO_BORDER)
        address = wx.ComboBox(toolbar1, -1, size=(50, -1))
        go = wx.BitmapButton(toolbar1, -1, wx.Bitmap('icons/go.png'),
                style=wx.NO_BORDER)
        text = wx.TextCtrl(toolbar1, -1, size=(150, -1))

        hbox1.Add(back)
        hbox1.Add(forward)
        hbox1.Add(refresh)
        hbox1.Add(stop)
        hbox1.Add(home)
        hbox1.Add(address, 1, wx.TOP, 4)
        hbox1.Add(go, 0, wx.TOP | wx.LEFT, 4)
        hbox1.Add(text, 0, wx.TOP | wx.RIGHT, 4)

        vbox.Add(toolbar1, 0, wx.EXPAND)
        line = wx.StaticLine(panel)
        vbox.Add(line, 0, wx.EXPAND)

        toolbar2 = wx.Panel(panel, -1, size=(-1, 30))
        bookmark1 = wx.BitmapButton(toolbar2, -1, wx.Bitmap('icons/love.png'),
                style=wx.NO_BORDER)
```

```
        bookmark2 = wx.BitmapButton(toolbar2, -1, wx.Bitmap('icons/books.png'),
                style=wx.NO_BORDER)
        bookmark3 = wx.BitmapButton(toolbar2, -1, wx.Bitmap('icons/sound.png'),
                style=wx.NO_BORDER)
        hbox2.Add(bookmark1, flag=wx.RIGHT, border=5)
        hbox2.Add(bookmark2, flag=wx.RIGHT, border=5)
        hbox2.Add(bookmark3)
        toolbar2.SetSizer(hbox2)
        vbox.Add(toolbar2, 0, wx.EXPAND)
        line = wx.StaticLine(panel)
        vbox.Add(line, 0, wx.EXPAND)

        panel.SetSizer(vbox)

        self.CreateStatusBar()
        self.Centre()
        self.Show(True)

app = wx.App(0)
Browser(None, -1, 'Browser')
app.MainLoop()
```

The question was, how to create a sizeable combo box, that is used in both Firefox and Opera? We cannot use a *wx.Toolbar*. It is not possible to create such a functionality with *wx.Toolbar*. Confirmed with Robin Dunn. So we must do a workaround.

```
toolbar1 = wx.Panel(panel, -1, size=(-1, 40))
```

The trick is simple. We create a plain *wx.Panel*.

```
hbox1 = wx.BoxSizer(wx.HORIZONTAL)
...
hbox1.Add(back)
hbox1.Add(forward)
hbox1.Add(refresh)
```

We create a horizontal sizer and add all necessary buttons.

```
hbox1.Add(address, 1, wx.TOP, 4)
```

Then we add the combo box to the sizer. This kind of combo box is usually called an address bar. Notice, that it is the only widget, that has the proportion set to 1. This was necessary to make it resizable.

The second toolbar was created in a similar way. The toolbars are separated by a line. First I thought, it was some kind of a panel border. I tested all possible borders, but it wasn't quite what I expected.

```
line = wx.StaticLine(panel)
```

Then I suddently got it. It is a simple static line!

Sometimes, we must create a solution, for which we don't have a

suitable widget. By using simple common sense, we can easily find a way.

Home ‡ Contents ‡ Top of Page

Creating custom widgets

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. wxPython 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.

Here I assume, you have read the chapter on the GDI.

A hyperlink widget

The first example will create a hyperlink. The hyperlink widget will be based on an existing *wx.lib.stattext.GenStaticText* widget.

```
#!/usr/bin/python

# link.py

import wx
from wx.lib.stattext import GenStaticText
import webbrowser



class Link(GenStaticText):
    def __init__(self, parent, id=-1, label='', pos=(-1, -1),
        size=(-1, -1), style=0, name='Link', URL=''):

        GenStaticText.__init__(self, parent, id, label, pos, size, style, name)

        self.url = URL
```

```
        self.font1 = wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, True, 'Verdana')
        self.font2 = wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, False, 'Verdana')

        self.SetFont(self.font2)
        self.SetForegroundColour('#0000ff')

        self.Bind(wx.EVT_MOUSE_EVENTS, self.OnMouseEvent)
        self.Bind(wx.EVT_MOTION, self.OnMouseEvent)


    def OnMouseEvent(self, event):
        if event.Moving():
            self.SetCursor(wx.StockCursor(wx.CURSOR_HAND))
            self.SetFont(self.font1)

        elif event.LeftUp():
            webbrowser.open_new(self.url)

        else:
            self.SetCursor(wx.NullCursor)
            self.SetFont(self.font2)

        event.Skip()


class HyperLink(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(220, 150))

        panel = wx.Panel(self, -1)
        Link(panel, -1, 'ZetCode', pos=(10, 60), URL='http://www.zetcode.com')
        motto = GenStaticText(panel, -1, 'Knowledge only matters', pos=(10, 30))
        motto.SetFont(wx.Font(9, wx.SWISS, wx.NORMAL, wx.BOLD, False, 'Verdana'))


        self.Centre()
        self.Show(True)

app = wx.App()
HyperLink(None, -1, 'A Hyperlink')
app.MainLoop()
```

This hyperlink widget is based on an existing widget. In this example we don't draw anything, we just use an existing widget, which we modify a bit.

```
from wx.lib.stattext import GenStaticText
import webbrowser
```

Here we import the base widget from which we derive our hyperlink widget and the webbrowser module. webbrowser module is a standard python module. We will use it to open links in a default browser.

```
self.SetFont(self.font2)
self.SetForegroundColour('#0000ff')
```

The idea behind creating a hyperlink widget is simple. We inherit from a base *wx.lib.stattext.GenStaticText* widget class. So we have a text

widget. Then we modify it a bit to make a hyperlink out of this text. We change the font and the colour of the text. Hyperlinks are usually blue.

```
if event.Moving():
    self.SetCursor(wx.StockCursor(wx.CURSOR_HAND))
    self.SetFont(self.font1)
```

If we hover a mouse pointer over the link, we change the font to underlined and also change the mouse pointer to a hand cursor.

```
elif event.LeftUp():
    webbrowser.open_new(self.url)
```

If we left click on the link, we open the link in a defaul browser.

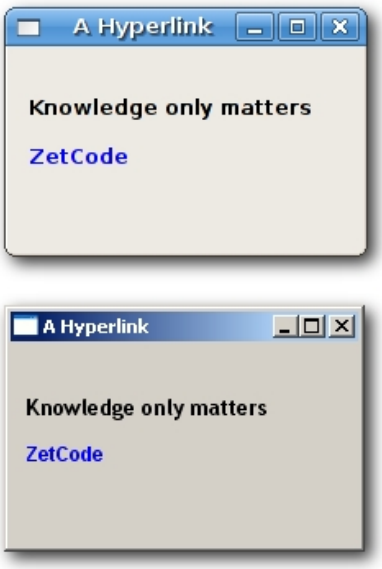


Figure: A Hyperlink widget

Burning widget

This is an example of a widget, that we create from a ground up. We put a **wx.Panel** on the bottom of the window and draw the entire widget manually. If you have ever burned a cd or a dvd, you already saw this kind of widget.

Remark for windows users. To avoid flicker, use double buffering.

```
#!/usr/bin/python

# burning.py

import wx
```

```
class Widget(wx.Panel):
    def __init__(self, parent, id):
        wx.Panel.__init__(self, parent, id, size=(-1, 30), style=wx.SUNKEN_BORDEI
        self.parent = parent
        self.font = wx.Font(9, wx.FONTFAMILY_DEFAULT, wx.FONTSTYLE_NORMAL,
            wx.FONTWEIGHT_NORMAL, False, 'Courier 10 Pitch')


        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_SIZE, self.OnSize)


    def OnPaint(self, event):
        num = range(75, 700, 75)
        dc = wx.PaintDC(self)
        dc.SetFont(self.font)
        w, h = self.GetSize()

        self.cw = self.parent.GetParent().cw

        step = int(round(w / 10.0))

        j = 0

        till = (w / 750.0) * self.cw
        full = (w / 750.0) * 700


        if self.cw >= 700:
            dc.SetPen(wx.Pen('#FFFFB8'))
            dc.SetBrush(wx.Brush('#FFFFB8'))
            dc.DrawRectangle(0, 0, full, 30)
            dc.SetPen(wx.Pen('#ffafaf'))
            dc.SetBrush(wx.Brush('#ffafaf'))
            dc.DrawRectangle(full, 0, till-full, 30)
        else:
            dc.SetPen(wx.Pen('#FFFFB8'))
            dc.SetBrush(wx.Brush('#FFFFB8'))
            dc.DrawRectangle(0, 0, till, 30)


        dc.SetPen(wx.Pen('#5C5142'))
        for i in range(step, 10*step, step):
            dc.DrawLine(i, 0, i, 6)
            width, height = dc.GetTextExtent(str(num[j]))
            dc.DrawText(str(num[j]), i-width/2, 8)
            j = j + 1

    def OnSize(self, event):
        self.Refresh()


class Burning(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(330, 200))

        self.cw = 75

        panel = wx.Panel(self, -1)
        CenterPanel = wx.Panel(panel, -1)
        self.sld = wx.Slider(CenterPanel, -1, 75, 0, 750, (-1, -1), (150, -1), w

        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox = wx.BoxSizer(wx.HORIZONTAL)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)
```

```
        hbox3 = wx.BoxSizer(wx.HORIZONTAL)

        self.wid = Widget(panel, -1)
        hbox.Add(self.wid, 1, wx.EXPAND)

        hbox2.Add(CenterPanel, 1, wx.EXPAND)
        hbox3.Add(self.sld, 0, wx.TOP, 35)

        CenterPanel.SetSizer(hbox3)

        vbox.Add(hbox2, 1, wx.EXPAND)
        vbox.Add(hbox, 0, wx.EXPAND)


        self.Bind(wx.EVT_SCROLL, self.OnScroll)

        panel.SetSizer(vbox)

        self.sld.SetFocus()

        self.Centre()
        self.Show(True)

    def OnScroll(self, event):
        self.cw = self.sld.GetValue()
        self.wid.Refresh()


app = wx.App()
Burning(None, -1, 'Burning widget')
app.MainLoop()
```

All the important code resides in the *OnPaint()* method of the Widget class. This widget shows graphically the total capacity of a medium and the free space available to us. The widget is controlled by a slider 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.

```
w, h = self.GetSize()
self.cw = self.parent.GetParent().cw
...
till = (w / 750.0) * self.cw
full = (w / 750.0) * 700
```

We draw the widget dynamically. The greater the window, the greater the burning widget. And vice versa. That is why we must calculate the size of the *wx.Panel* onto which we draw the custom widget. 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. Notice the use of floating point arithmetics. This is to achieve greater precision.

The actual drawing consists of three steps. We draw the yellow or red and yellow rectangle. Then we draw the vertical lines, which divide the widget into several parts. Finally, we draw the numbers, which indicate the capacity of the medium.

```
def OnSize(self, event):
    self.Refresh()
```

Every time the window is resized, we refresh the widget. This causes the widget to repaint itself.

```
def OnScroll(self, event):
    self.cw = self.sld.GetValue()
    self.wid.Refresh()
```

If we scroll the thumb of the slider, we get the actual value and save it into the *self.cw* parameter. This value is used, when the burning widget is drawn. Then we cause the widget to be redrawn.

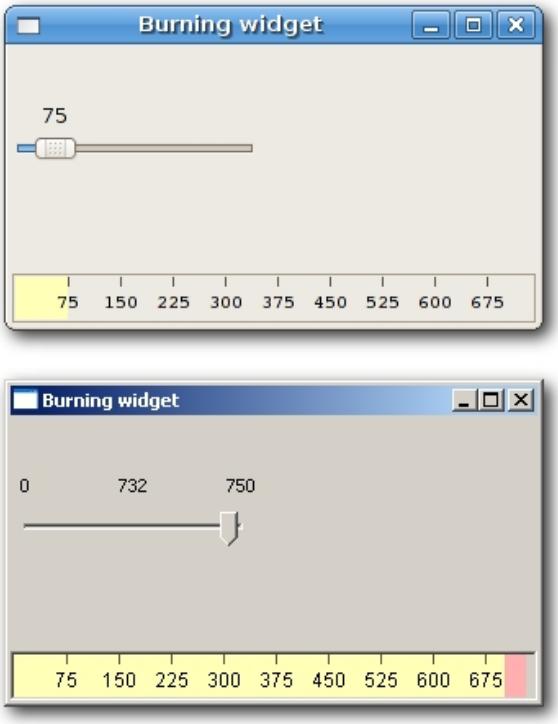


Figure: Burning widget

The CPU widget

There are system applications that measure system resources. The temperature, memory and CPU consuption etc. By displaying a simple text like CPU 54% you probably won't impress your users. Specialized widgets are created to make the application more appealing.

The following widget is often used in system applications.

Remark for windows users. To avoid flicker, use double buffering. Change the size of the application and the width of the slider.

```
#!/usr/bin/python

# cpu.py

import wx


class CPU(wx.Panel):
    def __init__(self, parent, id):
        wx.Panel.__init__(self, parent, id, size=(80, 110))

        self.parent = parent

        self.SetBackgroundColour('#000000')


        self.Bind(wx.EVT_PAINT, self.OnPaint)


    def OnPaint(self, event):

        dc = wx.PaintDC(self)

        dc.SetDeviceOrigin(0, 100)
        dc.SetAxisOrientation(True, True)

        pos = self.parent.GetParent().GetParent().sel
        rect = pos / 5

        for i in range(1, 21):
            if i > rect:
                dc.SetBrush(wx.Brush('#075100'))
                dc.DrawRectangle(10, i*4, 30, 5)
                dc.DrawRectangle(41, i*4, 30, 5)
            else:
                dc.SetBrush(wx.Brush('#36ff27'))
                dc.DrawRectangle(10, i*4, 30, 5)
                dc.DrawRectangle(41, i*4, 30, 5)


class CPUWidget(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(190, 140))

        self.sel = 0

        panel = wx.Panel(self, -1)
        centerPanel = wx.Panel(panel, -1)

        self.cpu = CPU(centerPanel, -1)

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        self.slider = wx.Slider(panel, -1, self.sel, 0, 100, (-1, -1), (25, 90),
                wx.VERTICAL | wx.SL_LABELS | wx.SL_INVERSE)
        self.slider.SetFocus()

        hbox.Add(centerPanel, 0,  wx.LEFT | wx.TOP, 20)
        hbox.Add(self.slider, 0, wx.LEFT | wx.TOP, 23)


        self.Bind(wx.EVT_SCROLL, self.OnScroll)

        panel.SetSizer(hbox)
```

```
        self.Centre()
        self.Show(True)


    def OnScroll(self, event):
        self.sel = event.GetInt()
        self.cpu.Refresh()


app = wx.App()
CPUWidget(None, -1, 'cpu')
app.MainLoop()
```

Creating this widget is quite simple. We create a black panel. Then we draw small rectangles onto this panel. The color of the rectangles depend on the value of the slider. The color can be dark green or bright green.

```
dc.SetDeviceOrigin(0, 100)
dc.SetAxisOrientation(True, True)
```

Here we change the default coordinate system to cartesian. This is to make the drawing intuitive.

```
pos = self.parent.GetParent().GetParent().sel
rect = pos / 5
```

Here we get the value of the sizer. We have 20 rectangles in each column. The slider has 100 numbers. The rect parameter makes a convertion from slider values into rectangles, that will be drawn in bright green color.

```
for i in range(1, 21):
    if i > rect:
        dc.SetBrush(wx.Brush('#075100'))
        dc.DrawRectangle(10, i*4, 30, 5)
        dc.DrawRectangle(41, i*4, 30, 5)
    else:
        dc.SetBrush(wx.Brush('#36ff27'))
        dc.DrawRectangle(10, i*4, 30, 5)
        dc.DrawRectangle(41, i*4, 30, 5)
```

Here we draw 40 rectangles, 20 in each column. If the number of the rectangle being drawn is greater than the converted rect value, we draw it in a dark green color. Otherwise in bright green.

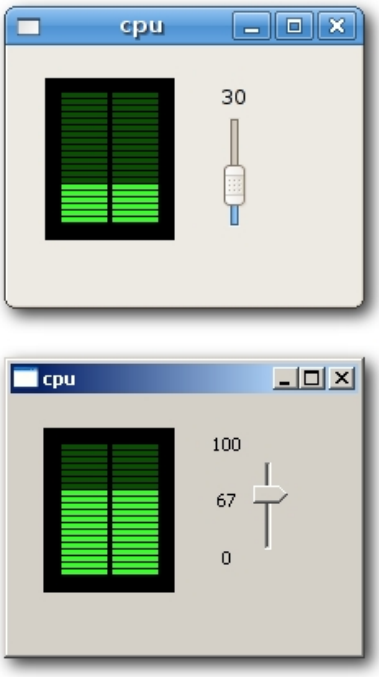


Figure: cpu widget

Home ‡ Contents ‡ Top of Page

xml resource files

The idea behind xml resources is to separate the interface from the code of an application. Several GUI builders use this concept for creating interfaces. For example the famous Glade. In our example we create a simple frame window with one button. We load resources from a file, load a panel and bind an event to a button.

Figure: myconfig.py

```
#!/usr/bin/python

# xml.py

import  wx
import  wx.xrc  as  xrc


class Xml(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        res = xrc.XmlResource('resource.xrc')
        res.LoadPanel(self, 'MyPanel')

        self.Bind(wx.EVT_BUTTON, self.OnClose,  id=xrc.XRCID('CloseButton'))
        self.Center()
        self.Show(True)


    def OnClose(self, event):
        self.Close()

app = wx.App()
Xml(None, -1, 'xml.py')
app.MainLoop()
```

This is resource file resource.xrc It is a xml file, where we define our widgets and their patterns. In thid file, we use tags like

```
<object></object>, <item></item>
```

etc.

```
<?xml version="1.0" ?>
<resource>
  <object class="wxPanel" name="MyPanel">
```

```
    <object class="wxButton" name="CloseButton">
     <label>Close</label>
     <pos>15,10</pos>
    </object>
   </object>
 </resource>
```

We use these two calls for working with widgets:

- XRCID(resource_name) - gives us the id of a button or menu item
- XRCCTRL(resource_name) - gives us the handlers of our widgets defined in resource file

Home ‡ Contents ‡ Top of Page

The GDI

The **GDI** (**Graphics Device Interface**) is an interface for working with graphics. It is used to interact with graphic devices such as monitor, printer or a file. The GDI allows programmers to display data on a screen or printer without having to be concerned about the details of a particular device. The GDI insulates the programmer from the hardware.

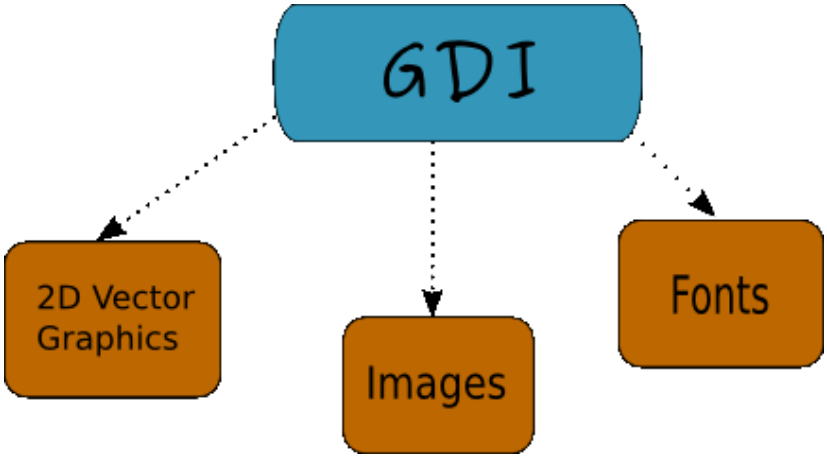


Figure: The GDI structure

From the programmer's point of view, the GDI is a group of classes and methods for working with graphics. The GDI consists of 2D Vector Graphics, Fonts and Images.

To begin drawing graphics, we must create a **device context** (**DC**) object. In wxPython the device context is called **wx.DC**. The documentation defines wx.DC as a device context onto which which graphics and text can be drawn. It represents number of devices in a generic way. Same piece of code can write to different kinds of devices. Be it a screen or a printer. The wx.DC is not intended to be used directly. Instead a programmer should choose one of the derived classes. Each derived class is intended to be used under specific conditions.

Derived wx.DC classes

- wxBufferedDC
- wxBufferedPaintDC
- wxPostScriptDC
- wxMemoryDC
- wxPrinterDC
- wxScreenDC
- wxClientDC
- wxPaintDC
- wxWindowDC

The *wx.ScreenDC* is used to draw anywhere on the screen. The *wx.WindowDC* is used if we want to paint on the whole window (Windows only). This includes window decorations. The *wx.ClientDC* is used to draw on the client area of a window. The client area is the area of a window without it's decorations (title and border). The *wx.PaintDC* is used to draw on the client area as well. But there is one difference between the *wx.PaintDC* and the *wx.ClientDC*. The *wx.PaintDC* should be used only from a *wx.PaintEvent*. The *wx.ClientDC* shoud not be used from a *wx.PaintEvent*. The *wx.MemoryDC* is used to draw graphics on the bitmap. The *wx.PostScriptDC* is used to write to PostScript files on any platform. The *wx.PrinterDC* is used to access a printer (Windows only).

Drawing a simple line

Our first example will draw a simple line onto the client area of a window.

```
DrawLine(int x1, int y1, int x2, int y2)
```

This method draws a line from the first point to the second. Excluding the second point.

```
#!/usr/bin/python

# line1.py

import wx

class Line(wx.Frame):
    def __init__(self, parent, id, title):
```

```
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        wx.FutureCall(2000, self.DrawLine)

        self.Centre()
        self.Show(True)

    def DrawLine(self):
        dc = wx.ClientDC(self)
        dc.DrawLine(50, 60, 190, 60)


app = wx.App()
Line(None, -1, 'Line')
app.MainLoop()
```

```
wx.FutureCall(2000, self.DrawLine)
```

We call the *DrawLine()* method after the window has been created. We do it because, when the window is created, it is drawn. All our drawings would be therefore lost. We can start drawing after the window has been created. This is the reason, why we call the *wx.FutureCall()* method.

```
def DrawLine(self):
    dc = wx.ClientDC(self)
    dc.DrawLine(50, 60, 190, 60)
```

We create a *wx.ClientDC* device context. The only parameter is the window on which we want to draw. In our case it is *self*, which is a reference to our *wx.Frame* widget. We call the *DrawLine()* method of the device context. This call actually draws a line on our window.

It is very important to understand the following behaviour. If we resize the window, the line will disappear. Why is this happening? Every window is redrawn, if it is resized. It is also redrawn, if it is maximized. The window is also redrawn, if we cover it by another window and uncover afterwards. The window is drawn to it's default state and our line is lost. We have to draw the line each time the window is resized. The solution is the *wx.PaintEvent*. This event is triggered every time, the window is redrawn. We will draw our line inside a method that will be hooked to the paint event.

The following example shows how it is done.

```
#!/usr/bin/python

# line2.py

import wx

class Line(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        dc.DrawLine(50, 60, 190, 60)

app = wx.App()
Line(None, -1, 'Line')
app.MainLoop()
```

```
self.Bind(wx.EVT_PAINT, self.OnPaint)
```

Here we bind the *OnPaint* method to the *wx.PaintEvent* event. It means, that each time our window is repainted, we call the *OnPaint* method. Now the line will not disappear, if we resize our window (cover it, maximize it).

```
dc = wx.PaintDC(self)
```

Notice, that this time we have used the *wx.PaintDC* device context.

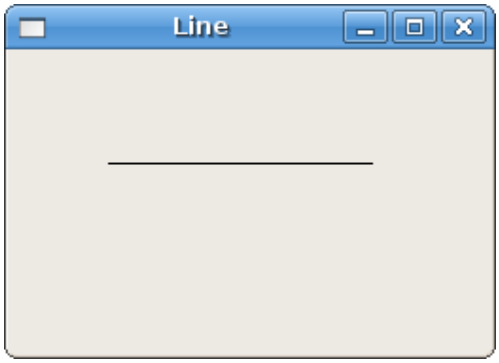


Figure: drawing a line

2D Vector Graphics

There are two different computer graphics. **Vector** and **raster** graphics. Raster graphics represents images as a collection of pixels. Vector graphics is the use of geometrical primitives such as points, lines, curves or polygons to represent images. These primitives are created using mathematical equations.

Both types of computer graphics have advantages and disadvantages. The advantages of vector graphics over raster are:

- smaller size
- ability to zoom indefinitely
- moving, scaling, filling or rotating does not degrade the quality of an image

Types of primitives

- points
- lines
- polylines
- polygons
- circles
- ellipses
- Splines

Device context attributes

Attribute	Object	Default value	Get Method
Brush	wx.Brush	wx.WHITE_BRUSH	wx.Brush GetBrush()
Pen	wx.Pen	wx.BLACK_PEN	wx.Pen GetPen()
Mapping Mode	-	wx.MM_TEXT	int GetMapMode()
BackgroundMode	-	wx.TRANSPARENT	int GetBackgroundMode()
Text background colour	wx.Colour	wx.WHITE	wx.Colour GetTextBackground()
Text foreground colour	wx.Colour	wx.BLACK	wx.Colour GetTextForeground()

BASIC ELEMENTS

In the following lines we will introduce several elementary objects. Colours, Brushes, Pens, Joins, Caps, Gradients.

Colours

A colour is an object representing a combination of Red, Green, and Blue (RGB) intensity values. Valid RGB values are in the range 0 to 255. There are three ways for setting colours. We can create a wx.Colour object, use a predefined colour name or use hex value string. *wx.Colour(0,0,255)*, *'BLUE'*, *'#0000FF'*. These three notations produce the same colour.

I prefer the hexadecimal notation. A perfect tool for working with colours can be found on the colorjack.com website. Or we can use such a tool as Gimp.

We have also a list of predefined colour names that we can use in our programs.

Standard Colour Database

AQUAMARINE	BLACK	BLUE	BLUE VIOLET	BROWN
CADET BLUE	CORAL	CORNFLOWER BLUE	CYAN	DARK GREY
DARK GREEN	DARK OLIVE GREEN	DARK ORCHID	DARK SLATE BLUE	DARK SLATE GREY
DARK TURQUOISE	DIM GREY	FIREBRICK	FOREST GREEN	GOLD
GOLDENROD	GREY	GREEN	GREEN YELLOW	INDIAN RED
KHAKI	LIGHT BLUE	LIGHT GREY	LIGHT STEEL BLUE	LIME GREEN
MAGENTA	MAROON	MEDIUM AQUAMARINE	MEDIUM BLUE	MEDIUM FOREST GREEN
MEDIUM GOLDENROD	MEDIUM ORCHID	MEDIUM SEA GREEN	MEDIUM SLATE BLUE	MEDIUM SPRING GREEN
MEDIUM TURQUOISE	MEDIUM VIOLET RED	MIDNIGHT BLUE	NAVY	ORANGE
ORANGE RED	ORCHID	PALE GREEN	PINK	PLUM
PURPLE	RED	SALMON	SEA GREEN	SIENNA
SKY BLUE	SLATE BLUE	SPRING GREEN	STEEL BLUE	TAN
THISTLE	TURQUOISE	VIOLET	VIOLET RED	WHEAT
WHITE	YELLOW	YELLOW GREEN		

```
#!/usr/bin/python

# colours.py

import wx

class Colours(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 280))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
```

```
        dc = wx.PaintDC(self)
        dc.SetPen(wx.Pen('#d4d4d4'))

        dc.SetBrush(wx.Brush('#c56c00'))
        dc.DrawRectangle(10, 15, 90, 60)

        dc.SetBrush(wx.Brush('#1ac500'))
        dc.DrawRectangle(130, 15, 90, 60)

        dc.SetBrush(wx.Brush('#539e47'))
        dc.DrawRectangle(250, 15, 90, 60)

        dc.SetBrush(wx.Brush('#004fc5'))
        dc.DrawRectangle(10, 105, 90, 60)

        dc.SetBrush(wx.Brush('#c50024'))
        dc.DrawRectangle(130, 105, 90, 60)

        dc.SetBrush(wx.Brush('#9e4757'))
        dc.DrawRectangle(250, 105, 90, 60)

        dc.SetBrush(wx.Brush('#5f3b00'))
        dc.DrawRectangle(10, 195, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c'))
        dc.DrawRectangle(130, 195, 90, 60)

        dc.SetBrush(wx.Brush('#785f36'))
        dc.DrawRectangle(250, 195, 90, 60)

app = wx.App()
Colours(None, -1, 'Colours')
app.MainLoop()
```

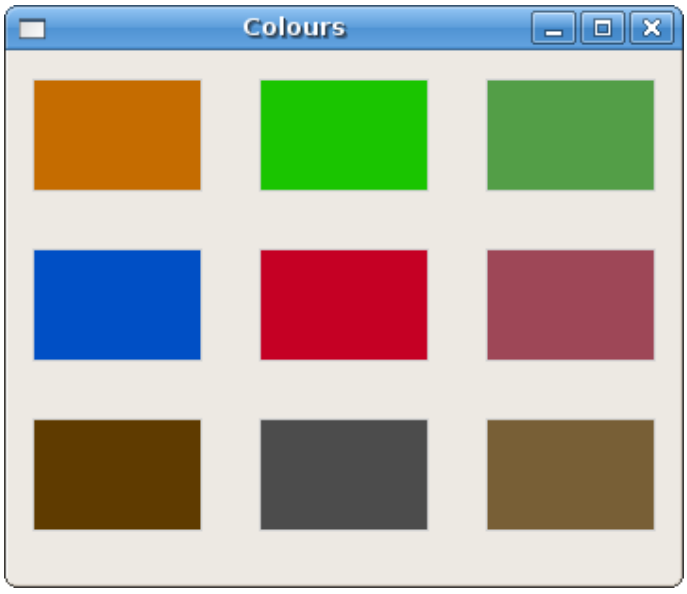


Figure: Colours

wx.Pen

Pen is an elementary graphics object. It is used to draw lines, curves and outlines of rectangles, ellipses, polygons or other shapes.

```
wx.Pen(wx.Colour colour, widht=1, style=wx.SOLID)
```

The *wx.Pen* constructor has three parameters. Colour, width and style. Follows a list of possible pen styles.

Pen styles

- wx.SOLID
- wx.DOT
- wx.LONG_DASH
- wx.SHORT_DASH
- wx.DOT_DASH
- wx.TRANSPARENT

```
#!/usr/bin/python

# pens.py

import wx

class Pens(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 190))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.SetPen(wx.Pen('#4c4c4c', 1, wx.SOLID))
        dc.DrawRectangle(10, 15, 90, 60)

        dc.SetPen(wx.Pen('#4c4c4c', 1, wx.DOT))
        dc.DrawRectangle(130, 15, 90, 60)

        dc.SetPen(wx.Pen('#4c4c4c', 1, wx.LONG_DASH))
        dc.DrawRectangle(250, 15, 90, 60)

        dc.SetPen(wx.Pen('#4c4c4c', 1, wx.SHORT_DASH))
```

```
        dc.DrawRectangle(10, 105, 90, 60)

        dc.SetPen(wx.Pen('#4c4c4c', 1, wx.DOT_DASH))
        dc.DrawRectangle(130, 105, 90, 60)

        dc.SetPen(wx.Pen('#4c4c4c', 1, wx.TRANSPARENT))
        dc.DrawRectangle(250, 105, 90, 60)

app = wx.App()
Pens(None, -1, 'Pens')
app.MainLoop()
```

If we don't specify a custom brush, a default one is used. The default brush is *wx.WHITE_BRUSH*. The perimeter of the rectangles is drawn by the pen. The last one has no border. It is transparent, e.g. not visible.

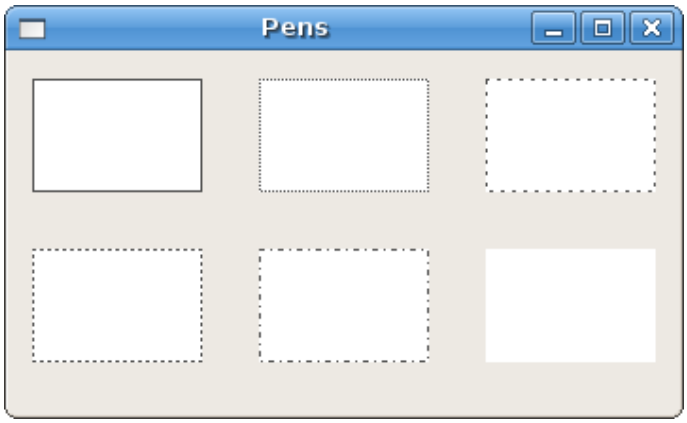


Figure: Pens

Joins and Caps

A pen object has additional two parameters. The *Join* and the *Cap*. The *Join* defines how joins between lines will be drawn. The *Join* style has the following options:

- wx.JOIN_MITER
- wx.JOIN_BEVEL
- wx.JOIN_ROUND

When using *wx.JOIN_MITER* the outer edges of the lines are extended. They meet at an angle, and this area is filled. In *wx.JOIN_BEVEL* the triangular notch between two lines is filled. In *wx.JOIN_ROUND* the circular arc between the two lines is filled. The default value is *wx.JOIN_ROUND*.

The *Cap* defines how the line ends will be drawn by the pen.

The options are:

- wx.CAP_ROUND
- wx.CAP_PROJECTING
- wx.CAP_BUTT

The *wx.CAP_ROUND* will draw rounded ends. The *wx.CAP_PROJECTING* and the *wx.CAP_BUTT* will both draw square ends. The difference between them is that the *wx.CAP_PROJECTING* will extend beyond the end point by the half of the line size. The *wx.CAP_ROUND* will extend beyond the end point as well.

```
#!/usr/bin/python

# joinscaps.py

import wx

class JoinsCaps(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(330, 300))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        pen = wx.Pen('#4c4c4c', 10, wx.SOLID)

        pen.SetJoin(wx.JOIN_MITER)
        dc.SetPen(pen)
        dc.DrawRectangle(15, 15, 80, 50)

        pen.SetJoin(wx.JOIN_BEVEL)
        dc.SetPen(pen)
        dc.DrawRectangle(125, 15, 80, 50)

        pen.SetJoin(wx.JOIN_ROUND)
        dc.SetPen(pen)
        dc.DrawRectangle(235, 15, 80, 50)

        pen.SetCap(wx.CAP_BUTT)
        dc.SetPen(pen)
        dc.DrawLine(30, 150,  150, 150)

        pen.SetCap(wx.CAP_PROJECTING)
        dc.SetPen(pen)
        dc.DrawLine(30, 190,  150, 190)
```

```
        pen.SetCap(wx.CAP_ROUND)
        dc.SetPen(pen)
        dc.DrawLine(30, 230,  150, 230)

        pen2 = wx.Pen('#4c4c4c', 1, wx.SOLID)
        dc.SetPen(pen2)
        dc.DrawLine(30, 130, 30, 250)
        dc.DrawLine(150, 130, 150, 250)
        dc.DrawLine(155, 130, 155, 250)

app = wx.App()
JoinsCaps(None, -1, 'Joins and Caps')
app.MainLoop()
```

```
pen = wx.Pen('#4c4c4c', 10, wx.SOLID)
```

In order to see the various *Join* and *Cap* styles, we need to set the pen width to be greater than 1.

```
dc.DrawLine(150, 130, 150, 250)
dc.DrawLine(155, 130, 155, 250)
```

Notice the two enclosing vertical lines. The distance between them is 5px. It is exactly the half of the current pen width.

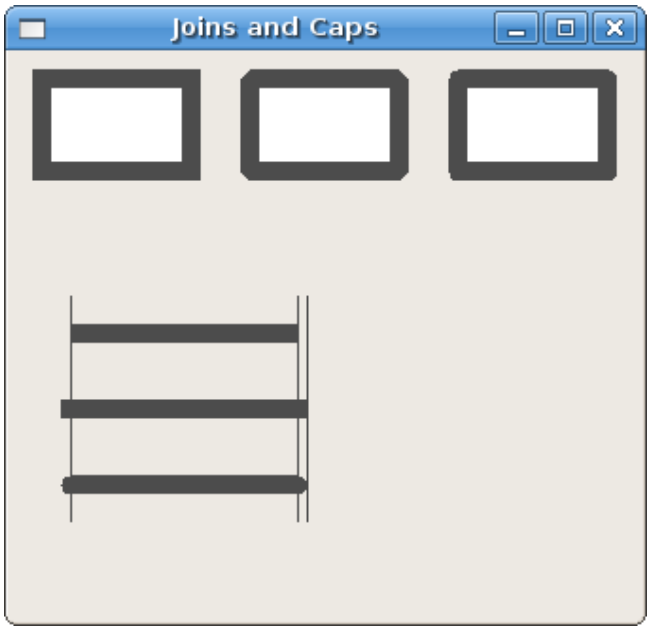


Figure: Joins and Caps

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)

```
GradientFillLinear(wx.Rect rect, wx.Colour initialColour, wx.Colour d
```

This method fills the area specified by a *rect* with a linear gradient, starting from *initialColour* and eventually fading to *destColour*. The nDirection parameter specifies the direction of the colour change, the default value is wx.EAST.

```
#!/usr/bin/python

# gradients.py

import wx

class Gradients(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(220, 260))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.GradientFillLinear((20, 20, 180, 40), '#ffec00', '#000000'
        dc.GradientFillLinear((20, 80, 180, 40), '#ffec00', '#000000'
        dc.GradientFillLinear((20, 140, 180, 40), '#ffec00', '#000000
        dc.GradientFillLinear((20, 200, 180, 40), '#ffec00', '#000000


app = wx.App()
Gradients(None, -1, 'Gradients')
app.MainLoop()
```

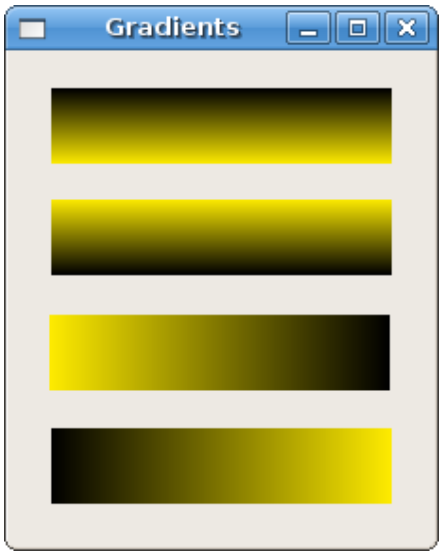


Figure: Gradients

wx.Brush

Brush is an elementary graphics object. It is used to paint the background of graphics shapes, such as rectangles, ellipses or polygons.

```
wx.Brush(wx.Colour colour, style=wx.SOLID)
```

The constructor of the *wx.Brush* accepts two parameters. Colour name and style. The following is a list of possible brush styles.

Brush styles

- wx.SOLID
- wx.STIPPLE
- wx.BDIAGONAL_HATCH
- wx.CROSSDIAG_HATCH
- wx.FDIAGONAL_HATCH
- wx.CROSS_HATCH
- wx.HORIZONTAL_HATCH
- wx.VERTICAL_HATCH
- wx.TRANSPARENT

```
#!/usr/bin/python

# brushes.py
```

```
import wx

class Brush(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 280))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.CROSS_HATCH))
        dc.DrawRectangle(10, 15, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.SOLID))
        dc.DrawRectangle(130, 15, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.BDIAGONAL_HATCH))
        dc.DrawRectangle(250, 15, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.CROSSDIAG_HATCH))
        dc.DrawRectangle(10, 105, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.FDIAGONAL_HATCH))
        dc.DrawRectangle(130, 105, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.HORIZONTAL_HATCH))
        dc.DrawRectangle(250, 105, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.VERTICAL_HATCH))
        dc.DrawRectangle(10, 195, 90, 60)

        dc.SetBrush(wx.Brush('#4c4c4c', wx.TRANSPARENT))
        dc.DrawRectangle(130, 195, 90, 60)


app = wx.App()
Brush(None, -1, 'Brushes')
app.MainLoop()
```

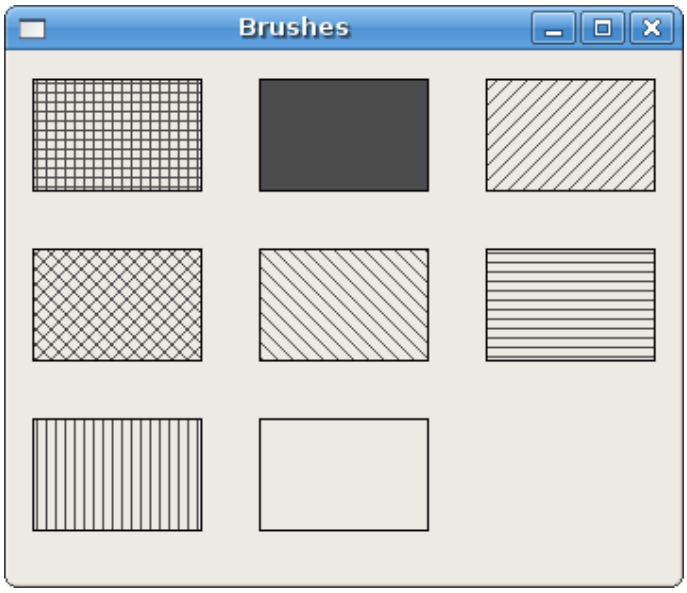


Figure: Brushes

Custom Patterns

We are not restricted to use predefined patterns. We can easily create our own custom patterns.

```
wx.Brush BrushFromBitmap(wx.Bitmap stippleBitmap)
```

This method creates a custom brush from the bitmap.

```
#!/usr/bin/python

# custompatterns.py

import wx

class CustomPatterns(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 280))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.SetPen(wx.Pen('#C7C3C3'))

        brush1 = wx.BrushFromBitmap(wx.Bitmap('pattern1.png'))
```

```
        dc.SetBrush(brush1)
        dc.DrawRectangle(10, 15, 90, 60)

        brush2 = wx.BrushFromBitmap(wx.Bitmap('pattern2.png'))
        dc.SetBrush(brush2)
        dc.DrawRectangle(130, 15, 90, 60)

        brush3 = wx.BrushFromBitmap(wx.Bitmap('pattern3.png'))
        dc.SetBrush(brush3)
        dc.DrawRectangle(250, 15, 90, 60)

        brush4 = wx.BrushFromBitmap(wx.Bitmap('pattern4.png'))
        dc.SetBrush(brush4)
        dc.DrawRectangle(10, 105, 90, 60)

        brush5 = wx.BrushFromBitmap(wx.Bitmap('pattern5.png'))
        dc.SetBrush(brush5)
        dc.DrawRectangle(130, 105, 90, 60)

        brush6 = wx.BrushFromBitmap(wx.Bitmap('pattern6.png'))
        dc.SetBrush(brush6)
        dc.DrawRectangle(250, 105, 90, 60)

        brush7 = wx.BrushFromBitmap(wx.Bitmap('pattern7.png'))
        dc.SetBrush(brush7)
        dc.DrawRectangle(10, 195, 90, 60)

        brush8 = wx.BrushFromBitmap(wx.Bitmap('pattern8.png'))
        dc.SetBrush(brush8)
        dc.DrawRectangle(130, 195, 90, 60)

        brushr9 = wx.BrushFromBitmap(wx.Bitmap('pattern9.png'))
        dc.SetBrush(brushr9)
        dc.DrawRectangle(250, 195, 90, 60)


app = wx.App()
CustomPatterns(None, -1, 'Custom Patterns')
app.MainLoop()
```

I have created some small bitmaps. For this I used the Gimp.

These bitmaps are rectangles, usually around 40-150px.

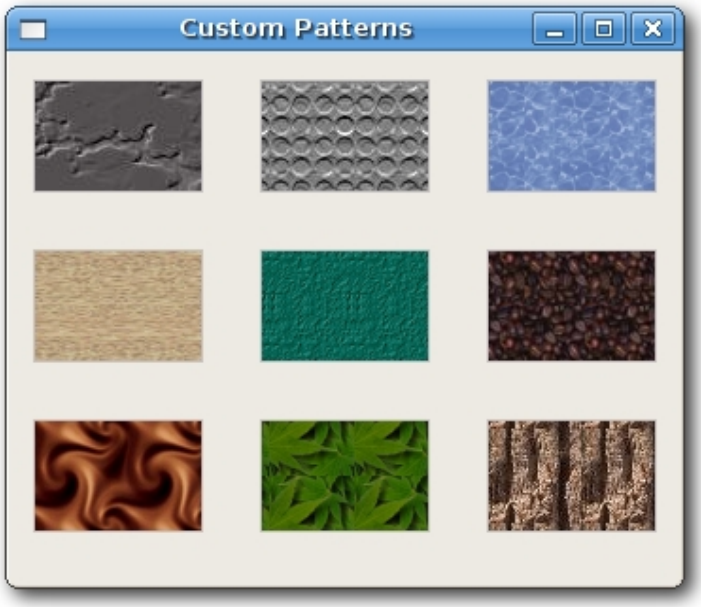


Figure: Custom Patterns

Basic primitives

Point

The simplest geometrical object is a point. It is a plain dot on the window.

```
DrawPoint(int x, int y)
```

This method draws a point at x, y coordinates.

```
#!/usr/bin/python

# points.py

import wx
import random

class Points(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)
```

```
    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.SetPen(wx.Pen('RED'))

        for i in range(1000):
            w, h = self.GetSize()
            x = random.randint(1, w-1)
            y = random.randint(1, h-1)
            dc.DrawPoint(x, y)


app = wx.App()
Points(None, -1, 'Points')
app.MainLoop()
```

A single point might be difficult to see. So we create 1000 points.

```
dc.SetPen(wx.Pen('RED'))
```

Here we set the colour of the pen to red.

```
w, h = self.GetSize()
x = random.randint(1, w-1)
```

The points are distributed randomly around the client area of the window. They are also distributed dynamically. If we resize the window, the points will be drawn randomly over a new client size. The *randint(a, b)* method returns a random integer in range [a, b], e.g. including both points.

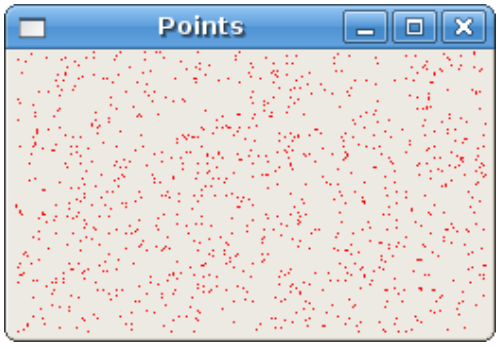


Figure: drawing points

Cross Hair

Cross Hair is a vertical and horizontal line the height and width of the window. It is centered on the given point.

```
CrossHair(int x, int y)
```

The method draws a cross hair centered on coordinates x, y.

```
#!/usr/bin/python

# crosshair.py

import wx

class CrossHair(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        dc.CrossHair(50, 50)


app = wx.App()
CrossHair(None, -1, 'CrossHair')
app.MainLoop()
```

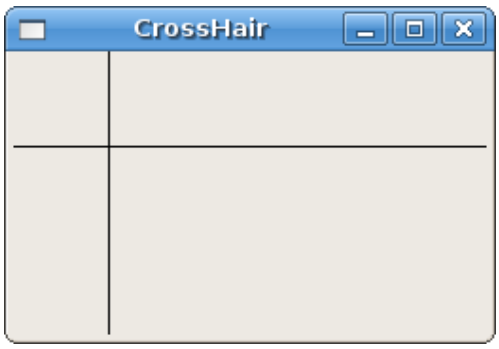


Figure: A Cross Hair

In the following code example we will create a functionality often seen in games. 'Aiming at the enemy.'

```
#!/usr/bin/python

# crosshair2.py
```

```
import wx

class CrossHair(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_MOTION, self.OnMotion)
        self.Bind(wx.EVT_LEAVE_WINDOW, self.OnLeaveWindow)

        self.SetBackgroundColour('WHITE')
        self.SetCursor(wx.StockCursor(wx.CURSOR_CROSS))
        self.Centre()
        self.Show(True)

    def DrawCrossHair(self, a, b):
        dc = wx.ClientDC(self)
        dc.Clear()
        dc.SetPen(wx.Pen(wx.Colour(100, 100, 100), 1, wx.DOT))
        dc.CrossHair(a, b)

    def OnMotion(self, event):
        x, y = event.GetPosition()
        self.DrawCrossHair(x, y)

    def OnLeaveWindow(self, event):
        dc = wx.ClientDC(self)
        dc.SetBackground(wx.Brush('WHITE'))
        dc.Clear()

app = wx.App()
CrossHair(None, -1, 'CrossHair2')
app.MainLoop()
```

```
self.Bind(wx.EVT_MOTION, self.OnMotion)
self.Bind(wx.EVT_LEAVE_WINDOW, self.OnLeaveWindow)
```

We bind two events to event handlers. The *wx.EVT_MOTION* event is generated, when we move a cursor over the window. The second event *wx.EVT_LEAVE_WINDOW* is generated, when we leave the window with our mouse cursor.

```
def OnMotion(self, event):
    x, y = event.GetPosition()
    self.DrawCrossHair(x, y)
```

Every time we move a cursor over a window, the method *OnMotion()* is called. In this method we figure out the current position of the mouse cursor and call the *DrawCrossHair* method, which is responsible for drawing the cross hair.

```
    def DrawCrossHair(self, a, b):
     dc = wx.ClientDC(self)
     dc.Clear()
     dc.SetPen(wx.Pen(wx.Colour(100, 100, 100), 1, wx.DOT))
     dc.CrossHair(a, b)
```

The user defined method *DrawCrossHair()* draws the cross hair primitive. Notice that to do the drawing, we use the *wx.ClientDC* device context. Remember that this device context should be used outside the paint event. This script is a perfect example, where we use *wx.ClientDC*. Not *wx.PaintDC*. Before we draw a new cross hair drawing, we must clear the old one. This is done with the *Clear()* method. It does clear the device context area. It uses the default *wx.WHITE_BRUSH* brush, unless we set a different one. We have set the window cursor to *wx.CURSOR_CROSS*. In order to actually see it, we have set the pen, which draws the cross hair, to light gray color, 1px wide, dotted.

Chech Mark

Check Mark is another simple primitive.

```
DrawCheckMark(int x, int y, int width, int height)
```

The *DrawCheckMark()* method draws a check mark on the window at coordinates x, y. It draws the check mark inside the rectangle defined by *width* and *height* parameters.

```
#!/usr/bin/python

# checkmark.py

import wx

class CheckMark(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 150))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)
```

```
    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        dc.DrawCheckMark(100, 50, 30, 40)


app = wx.App()
CheckMark(None, -1, 'Check Mark')
app.MainLoop()
```

Figure: Check Mark

SHAPES

Shapes are more sophisticated geometrical objects. We will draw various geometrical shapes in the following example.

```
#!/usr/bin/python

# shapes.py

import wx

class Shapes(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 300))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.DrawEllipse(20, 20, 90, 60)
        dc.DrawRoundedRectangle(130, 20, 90, 60, 10)
        dc.DrawArc(240, 40, 340, 40, 290, 20)
```

```
        dc.DrawPolygon(((130, 140), (180, 170), (180, 140), (220, 110
        dc.DrawRectangle(20, 120, 80, 50)
        dc.DrawSpline(((240, 170), (280, 170), (285, 110), (325, 110)

        dc.DrawLines(((20, 260), (100, 260), (20, 210), (100, 210)))
        dc.DrawCircle(170, 230, 35)
        dc.DrawRectangle(250, 200, 60, 60)

app = wx.App()
Shapes(None, -1, 'Shapes')
app.MainLoop()
```

In our example we have drawn an ellipse, a rounded rectangle, an arc, a rectangle ,a polygon, splines, lines, a circle and a square (from right to left, from top to bottom). A circle is a special kind of ellipse and a square is a special kind of rectangle.

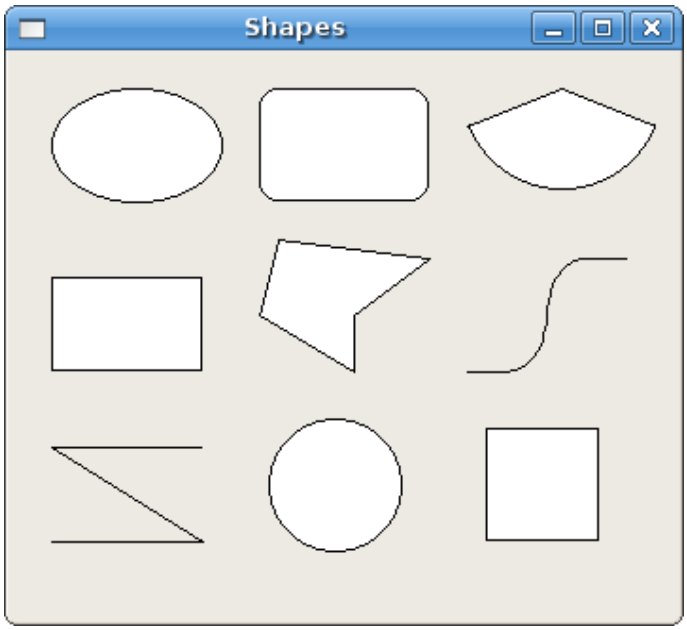


Figure: Shapes

REGIONS

The device context can be divided into several parts called **Regions**. A region can be of any shape. A region can be a simple rectangle or circle. With *Union*, *Intersect*, *Substract* and *Xor* operations we can create complex regions from simple ones. Regions are used for outlining, filling or clipping.

We can create regions in three ways. The easiest way is to create a rectangular region. More complex regions can be created from a list of points of from a bitmap.

```
wx.Region(int x=0, int y=0, int width=0, int height=0)
```

This constructor creates a rectangular region.

```
wx.RegionFromPoints(list points, int fillStyle=wx.WINDING_RULE)
```

This constructor creates a polygonal region. The *fillStyle* parameter can be wx.WINDING_RULE or wx.ODDEVEN_RULE.

```
wx.RegionFromBitmap(wx.Bitmap bmp)
```

The most complex regions can be created with the previous method.

Before we go to the regions, we will create a small example first. We divide the topic into several parts so that it is easier to understand. You may find it a good idea to revise your school math. Here we can find a good article.

```
#!/usr/bin/python

# lines.py

import wx
from math import hypot, sin, cos, pi

class Lines(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(450, 400))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        size_x, size_y = self.GetClientSizeTuple()
        dc.SetDeviceOrigin(size_x/2, size_y/2)

        radius = hypot(size_x/2, size_y/2)
```

```
        angle = 0

        while (angle < 2*pi):
            x = radius*cos(angle)
            y = radius*sin(angle)
            dc.DrawLinePoint((0, 0), (x, y))
            angle = angle + 2*pi/360

app = wx.App()
Lines(None, -1, 'Lines')
app.MainLoop()
```

In this example we draw 260 lines from the middle of the client area. The distance between two lines is 1 degree. We create an interesting figure.

```
import wx
from math import hypot, sin, cos, pi
```

We need three mathematical functions and one constant from the math module.

```
dc.SetDeviceOrigin(size_x/2, size_y/2)
```

The method *SetDeviceOrigin()* creates a new beginning of the coordinate system. We place it into the middle of the client area. By repositioning the coordinate system, we make our drawing less complicated.

```
radius = hypot(size_x/2, size_y/2)
```

Here we get the Hypotenuse. It is the longest line, we can draw from the middle of the client area. It is the length of the line, that should be drawn from the beginning into the corner of the window. This way most of the lines are not drawn fully. The overlapping parts are not visible. see Hypotenuse.

```
x = radius*cos(angle)
y = radius*sin(angle)
```

These are parametric functions. They are used to find [x, y] points on the curve. All 360 lines are drawn from the beginning of the coordinate system up to the points on the circle.

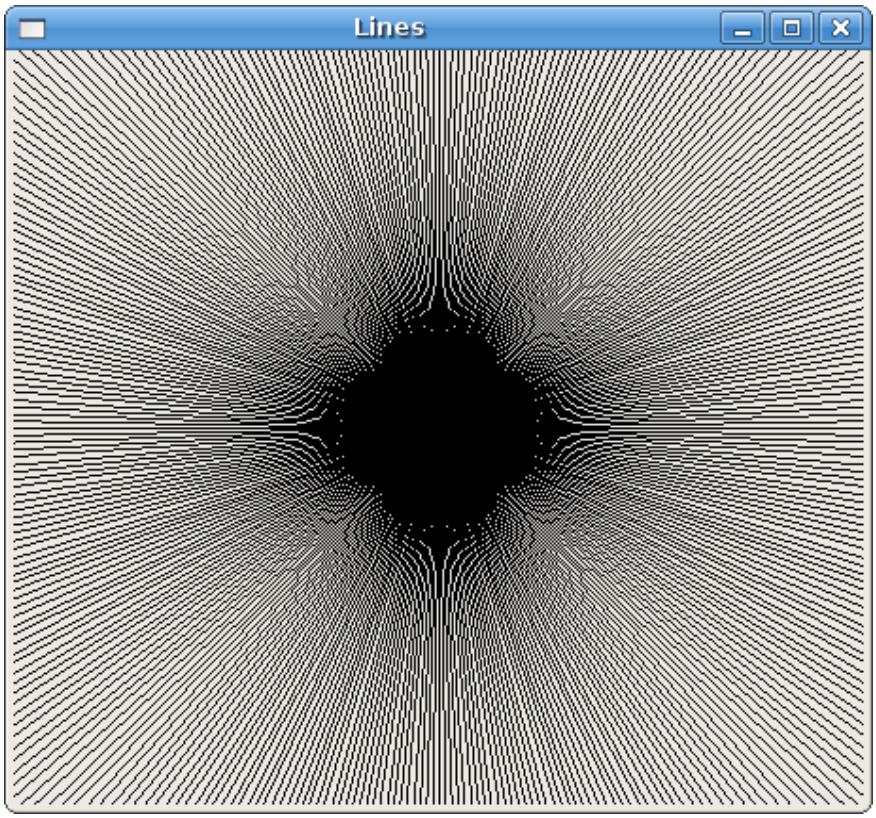


Figure: Lines

Clipping

Clipping is restricting drawing to a certain area. Clipping is used in two cases. To create effects and to improve performance of the application. We restrict drawing to a certain region with the *SetClippingRegionAsRegion()* method.

In the following example we will modify and enhance our previous script.

```
#!/usr/bin/python

# star.py

import wx
from math import hypot, sin, cos, pi

class Star(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(350, 300))

        self.Bind(wx.EVT_PAINT, self.OnPaint)

        self.Centre()
```

```
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        dc.SetPen(wx.Pen('#424242'))
        size_x, size_y = self.GetClientSizeTuple()
        dc.SetDeviceOrigin(size_x/2, size_y/2)

        points = (((0, 85), (75, 75), (100, 10), (125, 75), (200, 85)
            (150, 125), (160, 190), (100, 150), (40, 190), (50, 125))

        region = wx.RegionFromPoints(points)
        dc.SetClippingRegionAsRegion(region)

        radius = hypot(size_x/2, size_y/2)
        angle = 0

        while (angle < 2*pi):
            x = radius*cos(angle)
            y = radius*sin(angle)
            dc.DrawLinePoint((0, 0), (x, y))
            angle = angle + 2*pi/360

        dc.DestroyClippingRegion()


app = wx.App()
Star(None, -1, 'Star')
app.MainLoop()
```

We draw again all the 360 lines. But this time, only a portion of the client aren is drawn. The region that we restrict our drawing to is a star object.

```
region = wx.RegionFromPoints(points)
dc.SetClippingRegionAsRegion(region)
```

We create a region from the list of points with the *wx.RegionFromPoins()* method. The *SetClippingRegionAsRegion()* method restricts the drawing to the specified region. In our case it is a star object.

```
dc.DestroyClippingRegion()
```

We must destroy the clipping region.

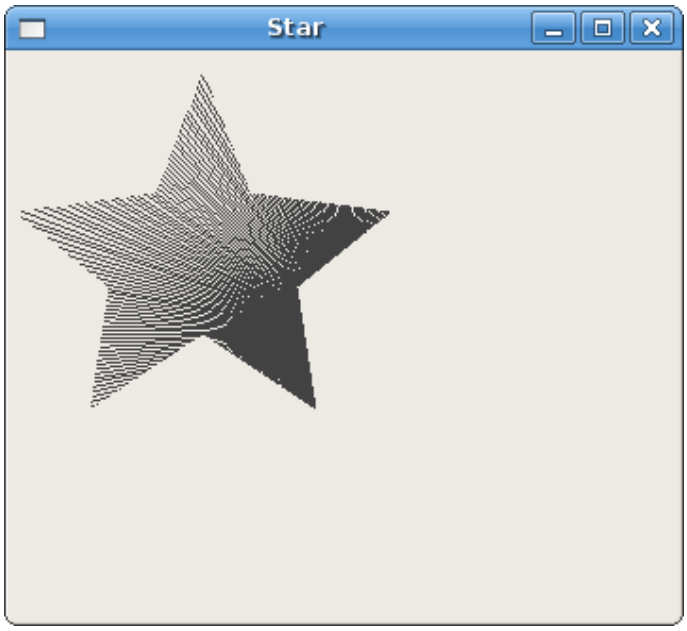


Figure: Star

Operations over Regions

Regions can be combined to create more complex shapes. We can use four set operations. *Union*, *Intersect*, *Substract* and *Xor*.

The following example shows all four operations in action.

```
#!/usr/bin/python

# operations.py

import wx

class Operations(wx.Frame):
     def __init__(self, parent, id, title):
         wx.Frame.__init__(self, parent, id, title, size=(270, 220))

         self.Bind(wx.EVT_PAINT, self.OnPaint)

         self.Centre()
         self.Show(True)

     def OnPaint(self, event):
         dc = wx.PaintDC(self)
         dc.SetPen(wx.Pen('#d4d4d4'))

         dc.DrawRectangle(20, 20, 50, 50)
         dc.DrawRectangle(30, 40, 50, 50)

         dc.SetBrush(wx.Brush('#ffffff'))
```

```
dc.DrawRectangle(100, 20, 50, 50)
dc.DrawRectangle(110, 40, 50, 50)
region1 = wx.Region(100, 20, 50, 50)
region2 = wx.Region(110, 40, 50, 50)
region1.IntersectRegion(region2)
rect = region1.GetBox()
dc.SetClippingRegionAsRegion(region1)
dc.SetBrush(wx.Brush('#ff0000'))
dc.DrawRectangleRect(rect)
dc.DestroyClippingRegion()

dc.SetBrush(wx.Brush('#ffffff'))
dc.DrawRectangle(180, 20, 50, 50)
dc.DrawRectangle(190, 40, 50, 50)
region1 = wx.Region(180, 20, 50, 50)
region2 = wx.Region(190, 40, 50, 50)
region1.UnionRegion(region2)
dc.SetClippingRegionAsRegion(region1)
rect = region1.GetBox()
dc.SetBrush(wx.Brush('#fa8e00'))
dc.DrawRectangleRect(rect)
dc.DestroyClippingRegion()

dc.SetBrush(wx.Brush('#ffffff'))
dc.DrawRectangle(20, 120, 50, 50)
dc.DrawRectangle(30, 140, 50, 50)
region1 = wx.Region(20, 120, 50, 50)
region2 = wx.Region(30, 140, 50, 50)
region1.XorRegion(region2)
rect = region1.GetBox()
dc.SetClippingRegionAsRegion(region1)
dc.SetBrush(wx.Brush('#619e1b'))
dc.DrawRectangleRect(rect)
dc.DestroyClippingRegion()

dc.SetBrush(wx.Brush('#ffffff'))
dc.DrawRectangle(100, 120, 50, 50)
dc.DrawRectangle(110, 140, 50, 50)
region1 = wx.Region(100, 120, 50, 50)
region2 = wx.Region(110, 140, 50, 50)
region1.SubtractRegion(region2)
rect = region1.GetBox()
dc.SetClippingRegionAsRegion(region1)
dc.SetBrush(wx.Brush('#715b33'))
dc.DrawRectangleRect(rect)
dc.DestroyClippingRegion()

dc.SetBrush(wx.Brush('#ffffff'))
dc.DrawRectangle(180, 120, 50, 50)
dc.DrawRectangle(190, 140, 50, 50)
region1 = wx.Region(180, 120, 50, 50)
region2 = wx.Region(190, 140, 50, 50)
region2.SubtractRegion(region1)
rect = region2.GetBox()
dc.SetClippingRegionAsRegion(region2)
dc.SetBrush(wx.Brush('#0d0060'))
```

```
            dc.DrawRectangleRect(rect)
            dc.DestroyClippingRegion()

app = wx.App()
Operations(None, -1, 'Operations')
app.MainLoop()
```

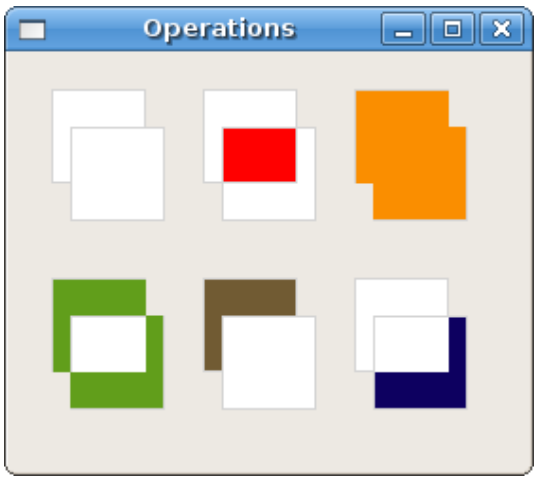


Figure: Set operations on Regions

MAPPING MODES

Speak in English, measure in Metric

The English language became the global language for communication. So did the metric system become the global system in measuremet. According to this wikipedia article, there are only three exceptions. The USA, Liberia and Myanmar. For example, Americans use Fahrenheits to measure temperature, gallons to tank their cars or pounds to weigh loads.

This might lead to some funny situations.
"It is very hot today. What is the temperature?"
"Let me see. It is one hundred and ,wait one hundred and ..."
?!?!
"... five Fahrenheits."
"Aha."

(When I was in US talking to an American)

Even though we in Europe use the metric system, there are

still exceptions. The USA is dominating the IT and we are importing their standards. So we also say that we have a 17 Inch monitor. Graphics can be put into a file, displayed on the screen of a monitor or other device (cameras, videocameras, mobile phones) or printed with a printer. Paper size can be set in millimeters, points or inches, the resolution of a screen is in pixels, the quality of a text is determined by the number of dots per inch. We have also dots, bits or samples. This is one of the reasons we have **logical** and **device** units.

Logical and device units

If we draw text or geometrical primitives on the client area, we position them using logical units.

```
DrawText(string text, int x, int y)
```

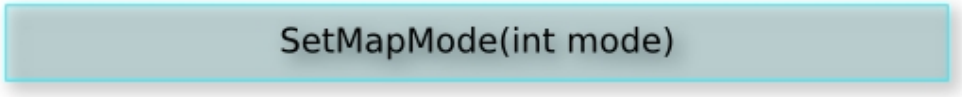
If we want to draw some text, we provide the text parameter and the x, y positions. x, y are in logical units. The device then draws the text in device units. Logical and device units may be the same, or they may differ. Logical units are used by people (millimeters), device units are are native to a *particular* device. For example a native device unit for a screen is pixel. The native device unit for the *HEWLETT PACKARD LaserJet 1022* is 1200 dpi. (dots per inch).

So far we have talked about various measurement units. The **mapping mode** of the device is a way how to convert logical units to device units. wxPython has the following mapping modes:

Mapping Mode	Logical Unit
wx.MM_TEXT	1 pixel
wx.MM_METRIC	1 millimeter
wx.MM_LOMETRIC	1/10 of a millimeter
wx.MM_POINTS	1 point, 1/72 of an inch
wx.MM_TWIPS	1/20 of a point or 1/1440 of an inch

The default mapping mode is wx.MM_TEXT. In this mode, the logical unit is the same as the device unit. When people position object on a screen or design a web page, they think usually in pixels. Web designers create three column pages

and these columns are set in pixels. The lowest common denominator for a page is often 800 px etc. This thinking is natural as we know our monitors have e.g. 1024x768 pxs. We are not going to do convertions, rather we are accustomed to think in pixels. If we want to draw a structure in millimeters, we can use the two metric mapping modes. Drawing directly in millimeters is too thick for a screen, that's why we have the wx.MM_LOMETRIC mapping mode.



To set a different mapping mode, we use the *SetMapMode()* method.

First ruler example

The first ruler example will measure screen objects in pixels.

```
#!/usr/bin/python

# ruler1.py

import wx


RW = 701 # ruler widht
RM = 10  # ruler margin
RH = 60  # ruler height


class Ruler1(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(RW + 2*RM, 6
            style=wx.FRAME_NO_TASKBAR | wx.NO_BORDER | wx.STAY_ON_TOP
        self.font = wx.Font(7, wx.FONTFAMILY_DEFAULT, wx.FONTSTYLE_NO
            wx.FONTWEIGHT_BOLD, False, 'Courier 10 Pitch')

        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_LEFT_DOWN, self.OnLeftDown)
        self.Bind(wx.EVT_RIGHT_DOWN, self.OnRightDown)
        self.Bind(wx.EVT_MOTION, self.OnMouseMove)

        self.Centre()
        self.Show(True)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)

        brush = wx.BrushFromBitmap(wx.Bitmap('granite.png'))
```

```
        dc.SetBrush(brush)
        dc.DrawRectangle(0, 0, RW+2*RM, RH)
        dc.SetFont(self.font)


        dc.SetPen(wx.Pen('#F8FF25'))
        dc.SetTextForeground('#F8FF25')


        for i in range(RW):
            if not (i % 100):
                dc.DrawLine(i+RM, 0, i+RM, 10)
                w, h = dc.GetTextExtent(str(i))
                dc.DrawText(str(i), i+RM-w/2, 11)
            elif not (i % 20):
                dc.DrawLine(i+RM, 0, i+RM, 8)
            elif not (i % 2): dc.DrawLine(i+RM, 0, i+RM, 4)

    def OnLeftDown(self, event):
        pos = event.GetPosition()
        x, y = self.ClientToScreen(event.GetPosition())
        ox, oy = self.GetPosition()
        dx = x - ox
        dy = y - oy
        self.delta = ((dx, dy))

    def OnMouseMove(self, event):
        if event.Dragging() and event.LeftIsDown():
            x, y = self.ClientToScreen(event.GetPosition())
            fp = (x - self.delta[0], y - self.delta[1])
            self.Move(fp)

    def OnRightDown(self, event):
        self.Close()

app = wx.App()
Ruler1(None, -1, '')
app.MainLoop()
```

In this example we create a ruler. This ruler will measure screen objects in pixels. We left the default mapping mode, which is *wx.MM_TEXT*. As we have already mentioned, this mode has the same logical and device units. In our case, pixels.

```
wx.Frame.__init__(self, parent, id, title, size=(RW + 2*RM, 60), style
        wx.NO_BORDER | wx.STAY_ON_TOP)
```

We have created a borderless window. The ruler is 721 px wide. The ruler is RW + 2*RM = 701 + 20 = 721. The ruler shows 700 numbers. 0 ... 700 is 701 pixels. A ruler has a

margin on both sides, 2*10 is 20 pixels. Together it makes 721 pixels.

```
brush = wx.BrushFromBitmap(wx.Bitmap('granite.png'))
dc.SetBrush(brush)
dc.DrawRectangle(0, 0, RW+2*RM, RH)
```

Here we draw a custom pattern onto the window. I have used a predefined pattern available in the GIMP. It is called granite.

```
w, h = dc.GetTextExtent(str(i))
dc.DrawText(str(i), i+RM-w/2, 11)
```

These lines ensure, that we align the text correctly. The *GetTextExtent()* method returns the width and the height of the text.

We do not have a border around our window. So we must handle moving manually by additional code. The *OnLeftDown()* and the *OnMouseMove()* methods enable us to move the ruler. (TODO:link to dragging.)

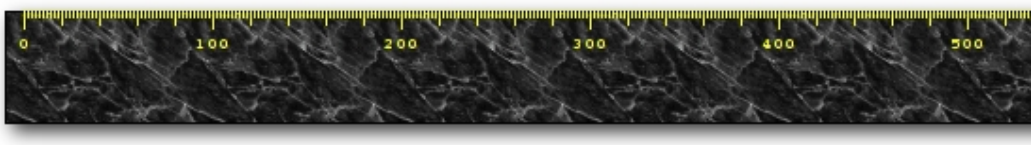


Figure: First ruler example

PRACTICAL EXAMPLES

You might ask yourself, why do we need all those lines, pens, gradients? What is it good for? The following scripts will bring some practical examples. We will utilize, what we have learnt in practice.

Charts

Creating charts is an excelent example of utilizing gdi drawing functions. Charts are not GUI widgets. No gui toolkit provides

charts as part of the library. One exception is wxWidgets toolkit (and so the wxPython). But these charts are very simple and cannot be used in real applications. A developer has usually two options. To create his own charting library or use a third-party library.

In the following example we create a simple line chart. We do not dwell into all details. I kept the example intentionally simple. A lot of things still remain undone. But you can grasp the idea and follow it.

```
#!/usr/bin/python

# linechart.py

import wx

data = ((10, 9), (20, 22), (30, 21), (40, 30), (50, 41),
(60, 53), (70, 45), (80, 20), (90, 19), (100, 22),
(110, 42), (120, 62), (130, 43), (140, 71), (150, 89),
(160, 65), (170, 126), (180, 187), (190, 128), (200, 125),
(210, 150), (220, 129), (230, 133), (240, 134), (250, 165),
(260, 132), (270, 130), (280, 159), (290, 163), (300, 94))

years = ('2003', '2004', '2005')


class LineChart(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent)
        self.SetBackgroundColour('WHITE')

        self.Bind(wx.EVT_PAINT, self.OnPaint)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        dc.SetDeviceOrigin(40, 240)
        dc.SetAxisOrientation(True, True)
        dc.SetPen(wx.Pen('WHITE'))
        dc.DrawRectangle(1, 1, 300, 200)
        self.DrawAxis(dc)
        self.DrawGrid(dc)
        self.DrawTitle(dc)
        self.DrawData(dc)

    def DrawAxis(self, dc):
        dc.SetPen(wx.Pen('#0AB1FF'))
        font =  dc.GetFont()
        font.SetPointSize(8)
        dc.SetFont(font)
        dc.DrawLine(1, 1, 300, 1)
        dc.DrawLine(1, 1, 1, 201)
```

```
        for i in range(20, 220, 20):
            dc.DrawText(str(i), -30, i+5)
            dc.DrawLine(2, i, -5, i)

        for i in range(100, 300, 100):
            dc.DrawLine(i, 2, i, -5)

        for i in range(3):
            dc.DrawText(years[i], i*100-13, -10)



    def DrawGrid(self, dc):
        dc.SetPen(wx.Pen('#d5d5d5'))

        for i in range(20, 220, 20):
            dc.DrawLine(2, i, 300, i)

        for i in range(100, 300, 100):
            dc.DrawLine(i, 2, i, 200)

    def DrawTitle(self, dc):
        font =  dc.GetFont()
        font.SetWeight(wx.FONTWEIGHT_BOLD)
        dc.SetFont(font)
        dc.DrawText('Historical Prices', 90, 235)


    def DrawData(self, dc):
        dc.SetPen(wx.Pen('#0ab1ff'))
        for i in range(10, 310, 10):
            dc.DrawSpline(data)


class LineChartExample(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(390, 300))

        panel = wx.Panel(self, -1)
        panel.SetBackgroundColour('WHITE')

        hbox = wx.BoxSizer(wx.HORIZONTAL)
        linechart = LineChart(panel)
        hbox.Add(linechart, 1, wx.EXPAND | wx.ALL, 15)
        panel.SetSizer(hbox)

        self.Centre()
        self.Show(True)


app = wx.App()
LineChartExample(None, -1, 'A line chart')
app.MainLoop()
```

```
dc.SetDeviceOrigin(40, 240)
dc.SetAxisOrientation(True, True)
```

By default the coordinate system in wxPython begins at point [0, 0]. The beginning point is located at the upper left corner of the clinet area. The orientation of x values is from left to right and the orientation of y values is from top to bottom. The values can be only positive. This system is used in all GUI toolkits. (All I am aware of.)

For charting we use cartesian coordinate system. In cartesian system, we can have both positive and negative values. The orientation of the x values is from left to right and the orientation of y values is from bottom to top. The origin is usually in the middle. But it is not compulsory.

```
dc.SetDeviceOrigin(40, 240)
dc.SetAxisOrientation(True, True)
```

The *SetDeviceOrigin()* method moves the origin to a new point on the client area. This is called **linear translation**. Then we change the axis orientation with the *SetAxisOrientation()* method.

```
SetAxisOrientation(bool xLeftRight, bool yBottomUp)
```

The method signature is self-explanatory. We can put true or false values to these two parameters.

```
self.DrawAxis(dc)
self.DrawGrid(dc)
self.DrawTitle(dc)
self.DrawData(dc)
```

We separate the construction of the chart into four methods. The first will draw axis, the second will draw the grid, the third the title and the last one will draw the data.

```
for i in range(3):
    dc.DrawText(years[i], i*100-13, -10)
```

Because of the simplicity of the script, there are some magic numbers. In reality, we would have to calculate them. In the previous code example, we draw the years alongside the x axis. We subtract 13 px from the x value. This is done to center the years over the vertical lines. It works on my linux box. I might not work correctly on other platforms. It might not work even on linux boxes with different themes. You just play a bit with this example. Adjusting it to fit under the different circumstances is no rocket science. Normally, we need to calculate the width of the chart, the width of the text and center the text manually.

Figure: A line chart

Note

Note is a small script that shows several interesting features of the GDI. We will see, how we can create a custom shaped window. There are small applications that are used to take visible notes. They work as reminders for people, that work with computers a lot. (e.g. us).

```
#!/usr/bin/python

# note.py

import wx
```

```
class Note(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title,
                          style=wx.FRAME_SHAPED |
                          wx.SIMPLE_BORDER |
                          wx.FRAME_NO_TASKBAR)

        self.font = wx.Font(11, wx.FONTFAMILY_DEFAULT, wx.FONTSTYLE_N
                wx.FONTWEIGHT_BOLD, False, 'Comic Sans MS')
        self.bitmap = wx.Bitmap('note.png', wx.BITMAP_TYPE_PNG)
        self.cross = wx.Bitmap('cross.png', wx.BITMAP_TYPE_PNG)

        w = self.bitmap.GetWidth()
        h = self.bitmap.GetHeight()
        self.SetClientSize((w, h))

        if wx.Platform == '__WXGTK__':
            self.Bind(wx.EVT_WINDOW_CREATE, self.SetNoteShape)
        else: self.SetNoteShape()

        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_LEFT_DOWN, self.OnLeftDown)
        self.Bind(wx.EVT_MOTION, self.OnMouseMove)

        self.bitmapRegion = wx.RegionFromBitmap(self.bitmap)
        self.crossRegion = wx.RegionFromBitmap(self.cross)

        self.bitmapRegion.IntersectRegion(self.crossRegion)
        self.bitmapRegion.Offset(170, 10)

        dc = wx.ClientDC(self)
        dc.DrawBitmap(self.bitmap, 0, 0, True)
        self.PositionTopRight()
        self.Show(True)

    def PositionTopRight(self):
        disx, disy = wx.GetDisplaySize()
        x, y = self.GetSize()
        self.Move((disx-x, 0))

    def SetNoteShape(self, *event):
        region = wx.RegionFromBitmap(self.bitmap)
        self.SetShape(region)

    def OnLeftDown(self, event):
        pos = event.GetPosition()
        if self.bitmapRegion.ContainsPoint(pos):
            self.Close()
        x, y = self.ClientToScreen(event.GetPosition())
        ox, oy = self.GetPosition()
        dx = x - ox
        dy = y - oy
        self.delta = ((dx, dy))
```

```
    def OnMouseMove(self, event):
        if event.Dragging() and event.LeftIsDown():
            x, y = self.ClientToScreen(event.GetPosition())
            fp = (x - self.delta[0], y - self.delta[1])
            self.Move(fp)

    def OnPaint(self, event):
        dc = wx.PaintDC(self)
        dc.SetFont(self.font)
        dc.SetTextForeground('WHITE')

        dc.DrawBitmap(self.bitmap, 0, 0, True)
        dc.DrawBitmap(self.cross, 170, 10, True)
        dc.DrawText('- Go shopping', 20, 20)
        dc.DrawText('- Make a phone call', 20, 50)
        dc.DrawText('- Write an email', 20, 80)


app = wx.App()
Note(None, -1, '')
app.MainLoop()
```

The idea behind creating a shaped window is simple. Most applications are rectangular. They share lots of similarities. They have menus, toolbars, titles etc. This might be boring. Some developers create more fancy applications. We can make our applications more attractive by using images. The idea is as follows. We create a frame without a border. We can draw a custom image on the frame during the paint event.

```
wx.Frame.__init__(self, parent, id, title,
                  style=wx.FRAME_SHAPED |
                  wx.SIMPLE_BORDER |
                  wx.FRAME_NO_TASKBAR)
```

In order to create a custom shaped application, we must set necessary style options. The *wx.FRAME_SHAPED* enables to create a shaped window. The *wx.SIMPLE_BORDER* removes the thick border. The *wx.FRAME_NO_TASKBAR* prevents the application from appearing on the taskbar.

```
self.font = wx.Font(11, wx.FONTFAMILY_DEFAULT, wx.FONTSTYLE_NORMAL,
                wx.FONTWEIGHT_BOLD, False, 'Comic Sans MS')
```

I was looking for a nice font for the note example. I finally chose Comic Sans MS. This is a proprietary font. Linux users must install msttcorefonts package. If we do not have a font

installed, the system chooses another one. Usually not to our taste.

```
self.bitmap = wx.Bitmap('note.png', wx.BITMAP_TYPE_PNG)
self.cross = wx.Bitmap('cross.png', wx.BITMAP_TYPE_PNG)
```

I have created two bitmaps. The first is a rounded rectangle. With a kind of an orange fill. I have used **Inkscape** vector illustrator to create it. The second one is a small cross. It is used to close the application. For this I used **Gimp** image editor.

```
w = self.bitmap.GetWidth()
h = self.bitmap.GetHeight()
self.SetClientSize((w, h))
```

We are going to draw a bitmap on the frame. I order to cover the whole frame, we figure out the bitmap size. Then we set the site of the frame to the size of the bitmap.

```
if wx.Platform == '__WXGTK__':
    self.Bind(wx.EVT_WINDOW_CREATE, self.SetNoteShape)
else: self.SetNoteShape()
```

This is some platform dependent code. Linux developers should call the *SetNoteShape()* method immediately after the *wx.WindowCreateEvent* event.

```
dc = wx.ClientDC(self)
dc.DrawBitmap(self.bitmap, 0, 0, True)
```

These lines are not necessary, because a paint event is generated during the creation of the application. But we believe, it makes the example smoother. I say we, because this is what I have learnt from the others.

```
def SetNoteShape(self, *event):
    region = wx.RegionFromBitmap(self.bitmap)
    self.SetShape(region)
```

Here we set the shape of the frame to that of the bitmap. The pixels outside the image become transparent.

If we remove a border from the frame, we cannot move the window. The *OnLeftDown()* and the *OnMouseMove()* methods enable the user to move the window by clicking on the client area of the frame and dragging it.

```
dc.DrawBitmap(self.bitmap, 0, 0, True)
dc.DrawBitmap(self.cross, 170, 10, True)
dc.DrawText('- Go shopping', 20, 20)
dc.DrawText('- Make a phone call', 20, 50)
dc.DrawText('- Write an email', 20, 80)
```

Within the *OnPaint()* method we draw two bitmaps and three texts.

Finally we will talk about how we close the note script.

```
self.bitmapRegion = wx.RegionFromBitmap(self.bitmap)
self.crossRegion = wx.RegionFromBitmap(self.cross)

self.bitmapRegion.IntersectRegion(self.crossRegion)
self.bitmapRegion.Offset(170, 10)
...
pos = event.GetPosition()
if self.bitmapRegion.ContainsPoint(pos):
    self.Close()
```

We create two regions from two bitmaps. We intersect these two regions. This way we get all pixels that share both bitmaps. Finally we move the region to the point, where we draw the cross bitmap. We use the *Offset()* method. By default the region starts at [0, 0] point.

Inside the *OnLeftDown()* method we check if we clicked inside the region. If true, we close the script.

Figure: Note

Home ‡ Contents ‡ Top of Page

Tips and Tricks

In this section we will show various interesting tips in wxPython. Here we will see examples, that could not be put elsewhere.

The tiniest wxPython application

This example is just for pure fun. Feel free to contact me, if you can shorthen it. Even for one single character. Except for the path to the python interpreter.

```
#!/usr/bin/python

import wx

i = wx.App()

wx.Frame(None).Show()

i.MainLoop()
```

Interactive Button

This tip shows how to program an interactive Button. This button reacts to users actions. In our case, the button changes it's background colour. When we enter the area of the button widget with a mouse pointer, wx/EVT_ENTER_WINDOW event is generated. Simirarly, wx.EVT_LEAVE_WINDOW event is generated, when we leave the area of the widget. So all you have to do is to bind those events to functions, that will change the colour/shape of the button widget appropriately.

```
#!/usr/bin/python

# interactivebutton.py
```

```
import wx
from wx.lib.buttons import GenButton

class InteractiveButton(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        panel = wx.Panel(self, -1)

        self.btn = GenButton(panel, -1, 'button', pos=(100, 100), siz
        self.btn.SetBezelWidth(1)
        self.btn.SetBackgroundColour( 'DARKGREY')

        wx.EVT_ENTER_WINDOW(self.btn, self.func)
        wx.EVT_LEAVE_WINDOW(self.btn, self.func1)

        self.Centre()
        self.Show(True)

    def func(self, event):
        self.btn.SetBackgroundColour( 'GREY79')
        self.btn.Refresh()

    def func1(self, event):
        self.btn.SetBackgroundColour( 'DARKGREY')
        self.btn.Refresh()


app = wx.App()
InteractiveButton(None, -1, 'interactivebutton.py')
app.MainLoop()
```

I have used a GenButton instead of a basic wx.Button. A GenButton can change border settigs, which I find attractive. But wx.Button would work as well.

Isabelle

When an error occurs in an application, an error dialog usually appears. This might get annoying. I have noticed a better solution in a SAP system. When a user enters an invalid command, statusbar turs red and an error message is displayed on stausbar. The red colour catches the eye and the user can easily read the error message. The following code mimics this situation.

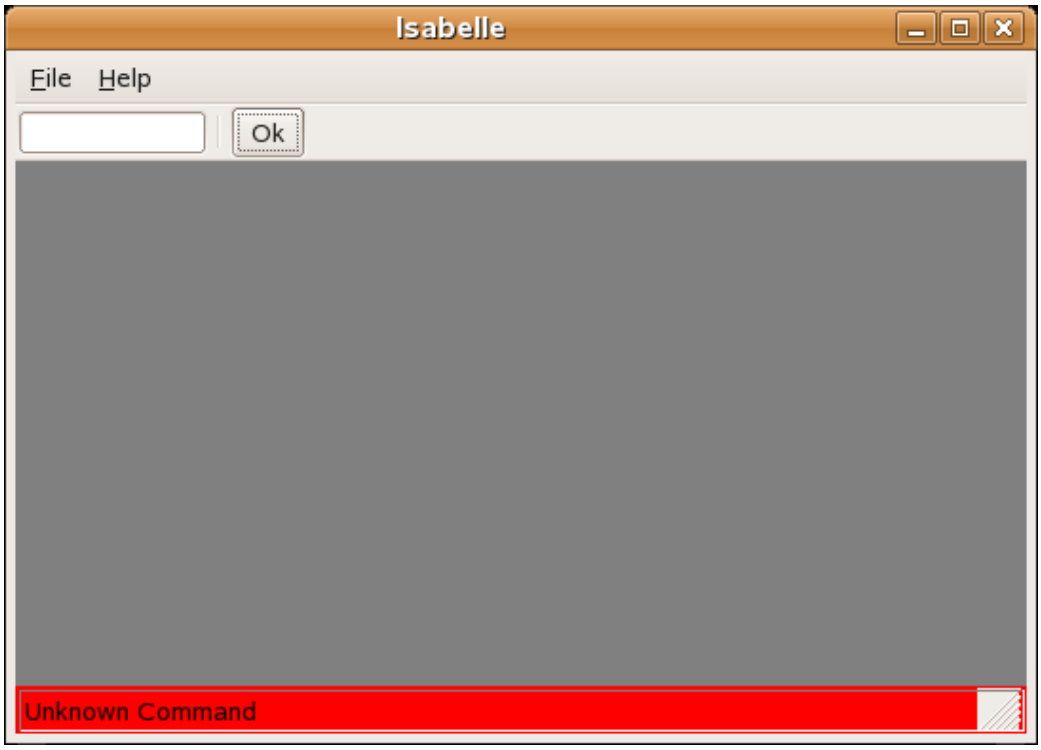


Figure: Isabelle

```
#!/usr/bin/python

# Isabelle

import wx

ID_TIMER = 1
ID_EXIT  = 2
ID_ABOUT = 3
ID_BUTTON = 4

class Isabelle(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title)

        self.timer = wx.Timer(self, ID_TIMER)
        self.blick = 0

        file = wx.Menu()
        file.Append(ID_EXIT, '&Quit\tCtrl+Q', 'Quit Isabelle')

        help = wx.Menu()
        help.Append(ID_ABOUT, '&About', 'O Programe')


        menubar = wx.MenuBar()
        menubar.Append(file, '&File')
        menubar.Append(help, '&Help')
        self.SetMenuBar(menubar)
```

```
        toolbar = wx.ToolBar(self, -1)
        self.tc = wx.TextCtrl(toolbar, -1, size=(100, -1))
        btn = wx.Button(toolbar, ID_BUTTON, 'Ok', size=(40, 28))

        toolbar.AddControl(self.tc)
        toolbar.AddSeparator()
        toolbar.AddControl(btn)
        toolbar.Realize()
        self.SetToolBar(toolbar)

        self.Bind(wx.EVT_BUTTON, self.OnLaunchCommandOk, id=ID_BUTTON
        self.Bind(wx.EVT_MENU, self.OnAbout, id=ID_ABOUT)
        self.Bind(wx.EVT_MENU, self.OnExit, id=ID_EXIT)
        self.Bind(wx.EVT_TIMER, self.OnTimer, id=ID_TIMER)

        self.panel = wx.Panel(self, -1, (0, 0), (500 , 300))
        self.panel.SetBackgroundColour('GRAY')
        self.sizer=wx.BoxSizer(wx.VERTICAL)
        self.sizer.Add(self.panel, 1, wx.EXPAND)
        self.SetSizer(self.sizer)
        self.statusbar = self.CreateStatusBar()
        self.statusbar.SetStatusText('Welcome to Isabelle')
        self.Centre()

    def OnExit(self, event):
        dlg = wx.MessageDialog(self, 'Are you sure to quit Isabelle?'
                wx.NO_DEFAULT | wx.ICON_QUESTION)
        if dlg.ShowModal() == wx.ID_YES:
            self.Close(True)


    def OnAbout(self, event):
        dlg = wx.MessageDialog(self, 'Isabelle\t\n' '2004\t', 'About'
                 wx.OK | wx.ICON_INFORMATION)
        dlg.ShowModal()
        dlg.Destroy()


    def OnLaunchCommandOk(self, event):
        input = self.tc.GetValue()
        if input == '/bye':
            self.OnExit(self)
        elif input == '/about':
            self.OnAbout(self)
        elif input == '/bell':
            wx.Bell()
        else:
            self.statusbar.SetBackgroundColour('RED')
            self.statusbar.SetStatusText('Unknown Command')
            self.statusbar.Refresh()
            self.timer.Start(50)

        self.tc.Clear()

    def OnTimer(self, event):
```

```
            self.blick = self.blick + 1
            if self.blick == 25:
                self.statusbar.SetBackgroundColour('#E0E2EB')
                self.statusbar.Refresh()
                self.timer.Stop()
                self.blick = 0

app = wx.App()
Isabelle(None, -1, 'Isabelle')
app.MainLoop()
```

There is a wx.TextCtrl on the Statusbar. There you enter your commands. We have defined three commands. /bye, /about and /beep. If you mistype any of them, Statusbar turns red and displays an error. This is done with the wx.Timer class.

Undo/Redo framework

Many applications have the ability to undo and redo the user's actions. The following example shows how it can be accomplished in wxPython.

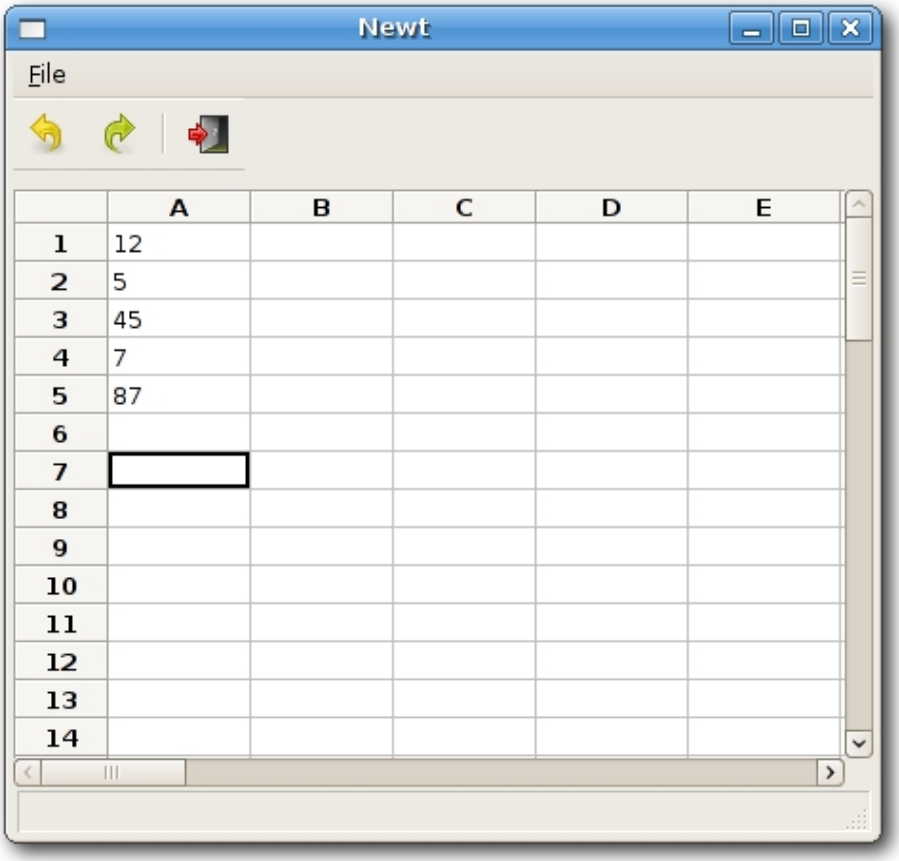


Figure: undoredo.py

```
#!/usr/bin/python

# undoredo.py

from wx.lib.sheet import *
import wx

stockUndo = []
stockRedo = []

ID_QUIT = 10
ID_UNDO = 11
ID_REDO = 12
ID_EXIT = 13

ID_COLSIZE = 80
ID_ROWSIZE = 20


class UndoText:
    def __init__(self, sheet, text1, text2, row, column):
        self.RedoText =  text2
        self.row = row
        self.col = column
        self.UndoText = text1
        self.sheet = sheet

    def undo(self):
        self.RedoText = self.sheet.GetCellValue(self.row, self.col)
        if self.UndoText ==  None:
            self.sheetSetCellValue('')
        else: self.sheet.SetCellValue(self.row, self.col, self.UndoTe

    def redo(self):
        if self.RedoText == None:
            self.sheet.SetCellValue('')
        else: self.sheet.SetCellValue(self.row, self.col, self.RedoTe

class UndoColSize:
    def __init__(self, sheet, position, size):
        self.sheet = sheet
        self.pos = position
        self.RedoSize = size
        self.UndoSize = ID_COLSIZE

    def undo(self):
        self.RedoSize = self.sheet.GetColSize(self.pos)
        self.sheet.SetColSize(self.pos, self.UndoSize)
        self.sheet.ForceRefresh()
```

```
    def redo(self):
        self.UndoSize = ID_COLSIZE
        self.sheet.SetColSize(self.pos, self.RedoSize)
        self.sheet.ForceRefresh()

class UndoRowSize:
    def __init__(self, sheet, position, size):
        self.sheet = sheet
        self.pos = position
        self.RedoSize = size
        self.UndoSize = ID_ROWSIZE

    def undo(self):
        self.RedoSize = self.sheet.GetRowSize(self.pos)
        self.sheet.SetRowSize(self.pos, self.UndoSize)
        self.sheet.ForceRefresh()

    def redo(self):
        self.UndoSize = ID_ROWSIZE
        self.sheet.SetRowSize(self.pos, self.RedoSize)
        self.sheet.ForceRefresh()

class MySheet(CSheet):
    instance = 0
    def __init__(self, parent):
        CSheet.__init__(self, parent)
        self.SetRowLabelAlignment(wx.ALIGN_CENTRE, wx.ALIGN_CENTRE)
        self.text = ''

    def OnCellChange(self, event):
        toolbar = self.GetParent().toolbar
        if (toolbar.GetToolEnabled(ID_UNDO) == False):
                toolbar.EnableTool(ID_UNDO, True)
        r = event.GetRow()
        c = event.GetCol()
        text = self.GetCellValue(r, c)
        # self.text - text before change
        # text - text after change
        undo = UndoText(self, self.text, text, r, c)
        stockUndo.append(undo)

        if stockRedo:
            # this might be surprising, but it is a standard behaviou
            # in all spreadsheets
            del stockRedo[:]
            toolbar.EnableTool(ID_REDO, False)

    def OnColSize(self, event):
        toolbar = self.GetParent().toolbar

        if (toolbar.GetToolEnabled(ID_UNDO) == False):
                toolbar.EnableTool(ID_UNDO, True)

        pos =  event.GetRowOrCol()
        size = self.GetColSize(pos)
        undo = UndoColSize(self, pos, size)
```

```
        stockUndo.append(undo)

        if stockRedo:
            del stockRedo[:]
            toolbar.EnableTool(ID_REDO, False)

    def OnRowSize(self, event):
        toolbar = self.GetParent().toolbar
        if (toolbar.GetToolEnabled(ID_UNDO) == False):
                toolbar.EnableTool(ID_UNDO, True)

        pos =  event.GetRowOrCol()
        size = self.GetRowSize(pos)
        undo = UndoRowSize(self, pos, size)

        stockUndo.append(undo)
        if stockRedo:
            del stockRedo[:]
            toolbar.EnableTool(ID_REDO, False)

class Newt(wx.Frame):
    def __init__(self,parent,id,title):
        wx.Frame.__init__(self, parent, -1, title, size=(550, 500))

        box = wx.BoxSizer(wx.VERTICAL)
        menuBar = wx.MenuBar()
        menu = wx.Menu()
        quit = wx.MenuItem(menu, ID_QUIT, '&Quit\tCtrl+Q', 'Quits New
        quit.SetBitmap(wx.Bitmap('icons/exit16.png'))
        menu.AppendItem(quit)
        menuBar.Append(menu, '&File')
        self.Bind(wx.EVT_MENU, self.OnQuitNewt, id=ID_QUIT)
        self.SetMenuBar(menuBar)


        self.toolbar = wx.ToolBar(self, id=-1, style=wx.TB_HORIZONTAL
                                        wx.TB_FLAT | wx.TB_TEXT)
        self.toolbar.AddSimpleTool(ID_UNDO, wx.Bitmap('icons/undo.png
              'Undo', '')
        self.toolbar.AddSimpleTool(ID_REDO, wx.Bitmap('icons/redo.png
              'Redo', '')
        self.toolbar.EnableTool(ID_UNDO, False)

        self.toolbar.EnableTool(ID_REDO, False)
        self.toolbar.AddSeparator()
        self.toolbar.AddSimpleTool(ID_EXIT, wx.Bitmap('icons/exit.png
              'Quit', '')
        self.toolbar.Realize()
        self.toolbar.Bind(wx.EVT_TOOL, self.OnUndo, id=ID_UNDO)
        self.toolbar.Bind(wx.EVT_TOOL, self.OnRedo, id=ID_REDO)
        self.toolbar.Bind(wx.EVT_TOOL, self.OnQuitNewt, id=ID_EXIT)

        box.Add(self.toolbar, border=5)
        box.Add((5,10), 0)

        self.SetSizer(box)
```

```
        self.sheet1 = MySheet(self)
        self.sheet1.SetNumberRows(55)
        self.sheet1.SetNumberCols(25)

        for i in range(self.sheet1.GetNumberRows()):
            self.sheet1.SetRowSize(i, ID_ROWSIZE)

        self.sheet1.SetFocus()
        box.Add(self.sheet1, 1, wx.EXPAND)
        self.CreateStatusBar()
        self.Centre()
        self.Show(True)

    def OnUndo(self, event):
        if len(stockUndo) == 0:
            return

        a = stockUndo.pop()
        if len(stockUndo) == 0:
            self.toolbar.EnableTool(ID_UNDO, False)

        a.undo()
        stockRedo.append(a)
        self.toolbar.EnableTool(ID_REDO, True)

    def OnRedo(self, event):
        if len(stockRedo) == 0:
            return

        a = stockRedo.pop()
        if len(stockRedo) == 0:
            self.toolbar.EnableTool(ID_REDO, False)

        a.redo()
        stockUndo.append(a)

        self.toolbar.EnableTool(ID_UNDO, True)

    def OnQuitNewt(self, event):
        self.Close(True)

app = wx.App()
Newt(None, -1, 'Newt')
app.MainLoop()
```

```
stockUndo = []
stockRedo = []
```

There are two list objects. stockUndo is a list that holds all changes, that we can undo. stockRedo keeps all changes, that can be redone. The changes are instantiated into a UndoText object. This object has two methods. undo and redo.

```
class MySheet(CSheet):
    def __init__(self, parent):
        CSheet.__init__(self, parent)
```

Our example inherits from CSheet class. It is a grid widget with some additional logic.

```
self.SetRowLabelAlignment(wx.ALIGN_CENTRE, wx.ALIGN_CENTRE)
```

Here we center the labels in rows. By default, they are aligned to the right.

```
r = event.GetRow()
c = event.GetCol()
text = self.GetCellValue(r, c)
# self.text - text before change
# text - text after change
undo = UndoText(self, self.text, text, r, c)
stockUndo.append(undo)
```

Every time we do some changes, an UndoText object is created and appended to the stockUndo list..

```
if stockRedo:
    # this might be surprising, but it is a standard behaviour
    # in all spreadsheets
    del stockRedo[:]
        toolbar.EnableTool(ID_REDO, False)
```

Yes, this behaviour was surprising for me. I did not know that it works this way, until I made this example. Basically, if you undo some changes and then start typing again, all redo changes are lost. OpenOffice Calc works this way. Gnumeric as well.

```
if len(stockUndo) == 0:
    self.toolbar.EnableTool(ID_UNDO, False)
...
self.toolbar.EnableTool(ID_REDO, True)
```

The undo and redo buttons are enabled or disabled

accordingly. If there is nothing to undo, the undo button is disabled.

```
a = stockUndo.pop()
if len(stockUndo) == 0:
    self.toolbar.EnableTool(ID_UNDO, False)

a.undo()
stockRedo.append(a)
```

If we click undo, we pop up an UndoText object from the stockUndo list. Call the *undo()* method and append the object to the stockRedo list.

Configuring application settings

Many applications allow users to configure their settings. Users can toggle tooltips on and of, change fonts, default download paths etc. Mostly they have a menu option called preferences. Application settings are saved to the hard disk, so that users do not have to change the settings each time the application starts.

In wxPython we have *wx.Config* class to do our job.

On Linux, settings are stored in a simple hidden file. This file is located in the home user directory by default. The location of the configuration file can be changed. The name of the file is specified in the constructor of the *wx.Config* class. In the following code example, we can cofigure the size of the window. If there is no configuration file, the height and the width of the window is set to the defaul 250 px value. We can set these values to a range from 200 - 500px. After we save our values and restart the application, the window size is set to our preffered values.

```
#!/usr/bin/python

# myconfig.py

import wx

class MyConfig(wx.Frame):
```

```
    def __init__(self, parent, id, title):
        self.cfg = wx.Config('myconfig')
        if self.cfg.Exists('width'):
            w, h = self.cfg.ReadInt('width'), self.cfg.ReadInt('heigh
        else:
            (w, h) = (250, 250)
        wx.Frame.__init__(self, parent, id, title, size=(w, h))

        wx.StaticText(self, -1, 'Width:', (20, 20))
        wx.StaticText(self, -1, 'Height:', (20, 70))
        self.sc1 = wx.SpinCtrl(self, -1, str(w), (80, 15), (60, -1),
        self.sc2 = wx.SpinCtrl(self, -1, str(h), (80, 65), (60, -1),
        wx.Button(self, 1, 'Save', (20, 120))

        self.Bind(wx.EVT_BUTTON, self.OnSave, id=1)
        self.statusbar = self.CreateStatusBar()
        self.Centre()
        self.Show(True)

    def OnSave(self, event):
        self.cfg.WriteInt("width", self.sc1.GetValue())
        self.cfg.WriteInt("height", self.sc2.GetValue())
        self.statusbar.SetStatusText('Configuration saved, %s ' % wx.


app = wx.App()
MyConfig(None, -1, 'myconfig.py')
app.MainLoop()
```

Here we have the contents of a configuration file to our code example . It consists of two key, value pairs.

```
$ cat .myconfig
height=230
width=350
```

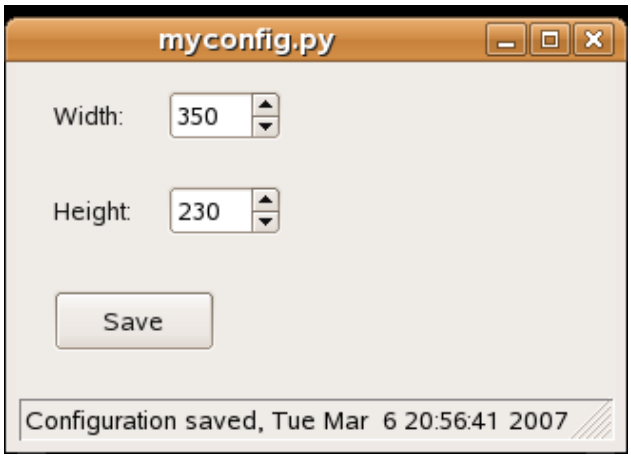


Figure: myconfig.py

Mouse gestures

A mouse gesture is a way of combining computer mouse movements and clicks which the software recognizes as a specific command. We can find mouse gestures in such successfull applications like Firefox or Opera. They really help users save their time while browsing on the Interent. Mouse gestures are created with wx.lib.gestures.MouseGestures class in wxPython.

Available gestures:

- L for left
- R for right
- U for up
- D for down
- 7 for northwest
- 9 for northeast
- 1 for southwest
- 3 for southeast

If you wonder why these numbers were chosen, have a look at the numerical pad. Mouse gestures can be combined. This way 'RDLU' is a mouse gesture triggered, when we do a square with a mouse pointer.

Possible flags are:

- wx.MOUSE_BTN_LEFT
- wx.MOUSE_BTN_MIDDLE
- wx.MOUSE_BTN_RIGHT

```
#!/usr/bin/python

# mousegestures.py

import wx
import wx.lib.gestures as gest

class MyMouseGestures(wx.Frame):

    def __init__ (self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(300, 200))

        panel = wx.Panel(self, -1)
```

```
        mg = gest.MouseGestures(panel, True, wx.MOUSE_BTN_LEFT)
        mg.SetGesturePen(wx.Colour(255, 0, 0), 2)
        mg.SetGesturesVisible(True)
        mg.AddGesture('DR', self.OnDownRight)

        self.Centre()
        self.Show(True)

    def OnDownRight(self):
          self.Close()

app = wx.App()
MyMouseGestures(None, -1, 'mousegestures.py')
app.MainLoop()
```

In our example, we have registered a mouse gesture for a panel. Mouse gesture is triggered, when a left button is pressed and we go down and right with a cursor. As in letter 'L'. Our mouse gesture will close the application.

```
mg = gest.MouseGestures(panel, True, wx.MOUSE_BTN_LEFT)
```

If we want to use mouse gestures, we have to create a MouseGesture object. The first parameter is a window, where the mouse gesture is registered. Second parameter defines a way to register a gesture. True is for automatic, False for manual. Manual is not fully implemented and we are happy with the automatic way. Last parameter defines a mouse button, which will be pressed when triggering gestures. The button can be later changed with the SetMouseButton() method.

```
mg.SetGesturePen(wx.Colour(255, 0, 0), 2)
```

Our gestures will be painted as red lines. They will be 2 pixels wide.

```
mg.SetGesturesVisible(True)
```

We set this gesture visible with the SetGesturesVisible() method.

```
mg.AddGesture('DR', self.OnDownRight)
```

We register a mouse gesture with the AddGesture() method.

The first parameter is the gesture. Second parameter is the method triggered by the gesture.

Home ‡ Contents ‡ Top of Page

wxPython Gripts

In this section we will show some small, complete scripts. These graphical scripts or "gripts" will demonstrate various areas in programming. Programming in Python, wxPython is easier than in most other toolkits. But it is still a laborious task. There is a long, long way from easy scripts to professional applications.

Tom

Each application should have a good name. Short and easily remembered. So, we have Tom. A simple gript that sends an email.

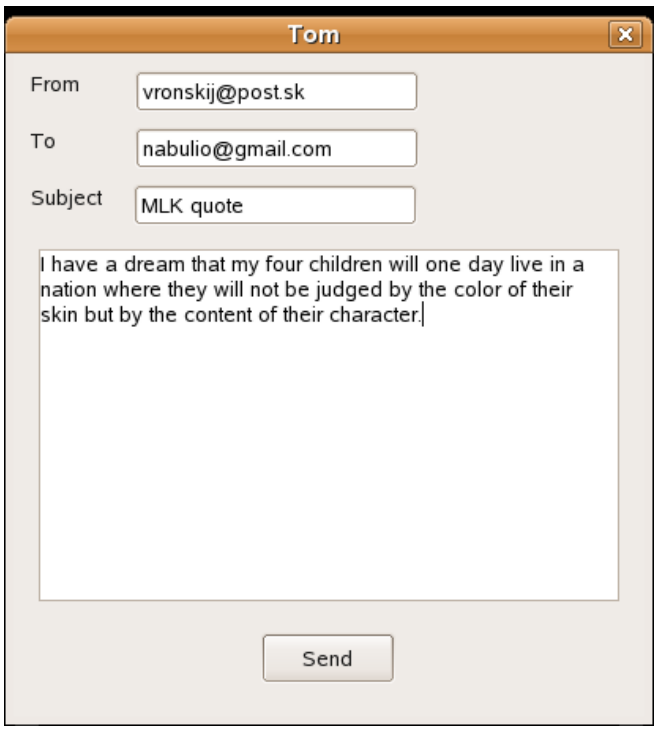


Figure: Tom

```
#!/usr/bin/python

# Tom

import wx
import smtplib

class Tom(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title, size=(400, 420))

        panel = wx.Panel(self, -1)
```

```
        vbox = wx.BoxSizer(wx.VERTICAL)
        hbox1 = wx.BoxSizer(wx.HORIZONTAL)
        hbox2 = wx.BoxSizer(wx.HORIZONTAL)
        hbox3 = wx.BoxSizer(wx.HORIZONTAL)

        st1 = wx.StaticText(panel, -1, 'From')
        st2 = wx.StaticText(panel, -1, 'To ')
        st3 = wx.StaticText(panel, -1, 'Subject')

        self.tc1 = wx.TextCtrl(panel, -1, size=(180, -1))
        self.tc2 = wx.TextCtrl(panel, -1, size=(180, -1))
        self.tc3 = wx.TextCtrl(panel, -1, size=(180, -1))

        self.write = wx.TextCtrl(panel, -1, style=wx.TE_MULTILINE)
        button_send = wx.Button(panel, 1, 'Send')

        hbox1.Add(st1, 0, wx.LEFT, 10)
        hbox1.Add(self.tc1, 0, wx.LEFT, 35)
        hbox2.Add(st2, 0, wx.LEFT, 10)
        hbox2.Add(self.tc2, 0, wx.LEFT, 50)
        hbox3.Add(st3, 0, wx.LEFT, 10)
        hbox3.Add(self.tc3, 0, wx.LEFT, 20)
        vbox.Add(hbox1, 0, wx.TOP, 10)
        vbox.Add(hbox2, 0, wx.TOP, 10)
        vbox.Add(hbox3, 0, wx.TOP, 10)
        vbox.Add(self.write, 1, wx.EXPAND | wx.TOP | wx.RIGHT | wx.LEFT, 15)
        vbox.Add(button_send, 0, wx.ALIGN_CENTER | wx.TOP | wx.BOTTOM, 20)

        self.Bind(wx.EVT_BUTTON, self.OnSend, id=1)
        panel.SetSizer(vbox)

        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnSend(self, event):
        sender = self.tc1.GetValue()
        recipient = self.tc2.GetValue()
        subject = self.tc3.GetValue()
        text = self.write.GetValue()
        header = 'From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n' % (sender, recipient, su
        message = header + text

        try:
            server = smtplib.SMTP('mail.chello.sk')
            server.sendmail(sender, recipient, message)
            server.quit()
            dlg = wx.MessageDialog(self, 'Email was successfully sent', 'Success',
                wx.OK | wx.ICON_INFORMATION)
            dlg.ShowModal()
            dlg.Destroy()

        except smtplib.SMTPException, error:
            dlg = wx.MessageDialog(self, 'Failed to send email', 'Error', wx.OK | wx.
            dlg.ShowModal()
            dlg.Destroy()


app = wx.App()
Tom(None, -1, 'Tom')
app.MainLoop()
```

For working with emails we need to import smtp module. This module is part of the python language.

```
import smtplib
```

From, To and Subject options must be separated by carriedge return and newline as shown here. This weird thing is requested by RFC 821 norm. So we must follow it.

```
header = 'From: %s\r\nTo: %s\r\nSubject: %s\r\n\r\n' % (sender, recipient, subject)
```

Next we create an SMTP connection. Here you specify your settings. Each ISP gives you the name of the pop and smtp servers. In my case, 'mail.chello.sk' is a name for both. A mail is sent by calling the sendmail() method. Finally, we quit the connection with the quit() method.

```
server = smtplib.SMTP('mail.chello.sk')
server.sendmail(sender, recipient, message)
server.quit()
```

Editor

This editor example is the largest so far.

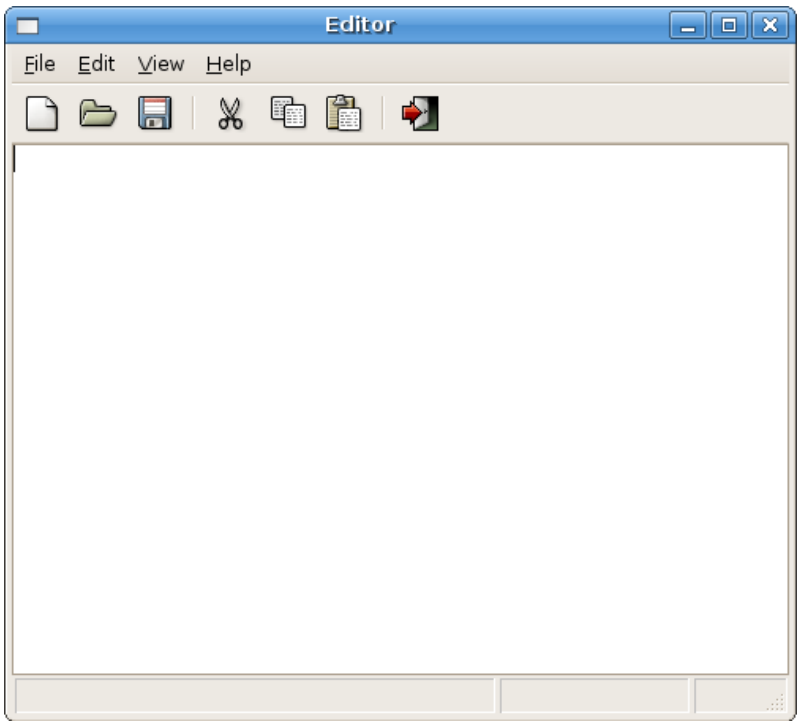


Figure: Editor

```
#!/usr/bin/python

# Editor

import wx
import os
```

```
class Editor(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(600, 500))

        # variables
        self.modify = False
        self.last_name_saved = ''
        self.replace = False

        # setting up menubar
        menubar = wx.MenuBar()

        file = wx.Menu()
        new = wx.MenuItem(file, 101, '&New\tCtrl+N', 'Creates a new document')
        new.SetBitmap(wx.Bitmap('icons/stock_new-16.png'))
        file.AppendItem(new)

        open = wx.MenuItem(file, 102, '&Open\tCtrl+O', 'Open an existing file')
        open.SetBitmap(wx.Bitmap('icons/stock_open-16.png'))
        file.AppendItem(open)
        file.AppendSeparator()

        save = wx.MenuItem(file, 103, '&Save\tCtrl+S', 'Save the file')
        save.SetBitmap(wx.Bitmap('icons/stock_save-16.png'))
        file.AppendItem(save)

        saveas = wx.MenuItem(file, 104, 'Save &As...\tShift+Ctrl+S',
                'Save the file with a different name')
        saveas.SetBitmap(wx.Bitmap('icons/stock_save_as-16.png'))
        file.AppendItem(saveas)
        file.AppendSeparator()

        quit = wx.MenuItem(file, 105, '&Quit\tCtrl+Q', 'Quit the Application')
        quit.SetBitmap(wx.Bitmap('icons/stock_exit-16.png'))
        file.AppendItem(quit)

        edit = wx.Menu()
        cut = wx.MenuItem(edit, 106, '&Cut\tCtrl+X', 'Cut the Selection')
        cut.SetBitmap(wx.Bitmap('icons/stock_cut-16.png'))
        edit.AppendItem(cut)

        copy = wx.MenuItem(edit, 107, '&Copy\tCtrl+C', 'Copy the Selection')
        copy.SetBitmap(wx.Bitmap('icons/stock_copy-16.png'))
        edit.AppendItem(copy)

        paste = wx.MenuItem(edit, 108, '&Paste\tCtrl+V', 'Paste text from clipboard')
        paste.SetBitmap(wx.Bitmap('icons/stock_paste-16.png'))
        edit.AppendItem(paste)

        delete = wx.MenuItem(edit, 109, '&Delete', 'Delete the selected text')
        delete.SetBitmap(wx.Bitmap('icons/stock_delete-16.png',))

        edit.AppendItem(delete)
        edit.AppendSeparator()
        edit.Append(110, 'Select &All\tCtrl+A', 'Select the entire text')

        view = wx.Menu()
        view.Append(111, '&Statusbar', 'Show StatusBar')

        help = wx.Menu()
        about = wx.MenuItem(help, 112, '&About\tF1', 'About Editor')
        about.SetBitmap(wx.Bitmap('icons/stock_about-16.png'))
        help.AppendItem(about)

        menubar.Append(file, '&File')
        menubar.Append(edit, '&Edit')
```

```
        menubar.Append(view, '&View')
        menubar.Append(help, '&Help')
        self.SetMenuBar(menubar)

        self.Bind(wx.EVT_MENU, self.NewApplication, id=101)
        self.Bind(wx.EVT_MENU, self.OnOpenFile, id=102)
        self.Bind(wx.EVT_MENU, self.OnSaveFile, id=103)
        self.Bind(wx.EVT_MENU, self.OnSaveAsFile, id=104)
        self.Bind(wx.EVT_MENU, self.QuitApplication, id=105)
        self.Bind(wx.EVT_MENU, self.OnCut, id=106)
        self.Bind(wx.EVT_MENU, self.OnCopy, id=107)
        self.Bind(wx.EVT_MENU, self.OnPaste, id=108)
        self.Bind(wx.EVT_MENU, self.OnDelete, id=109)
        self.Bind(wx.EVT_MENU, self.OnSelectAll, id=110)
        self.Bind(wx.EVT_MENU, self.ToggleStatusBar, id=111)
        self.Bind(wx.EVT_MENU, self.OnAbout, id=112)

        # setting up toolbar
        self.toolbar = self.CreateToolBar( wx.TB_HORIZONTAL | wx.NO_BORDER | wx.TB_FL
                | wx.TB_TEXT )
        self.toolbar.AddSimpleTool(801, wx.Bitmap('icons/stock_new.png'), 'New', '')
        self.toolbar.AddSimpleTool(802, wx.Bitmap('icons/stock_open.png'), 'Open', ''
        self.toolbar.AddSimpleTool(803, wx.Bitmap('icons/stock_save.png'), 'Save', ''
        self.toolbar.AddSeparator()

        self.toolbar.AddSimpleTool(804, wx.Bitmap('icons/stock_cut.png'), 'Cut', '')
        self.toolbar.AddSimpleTool(805, wx.Bitmap('icons/stock_copy.png'), 'Copy', ''
        self.toolbar.AddSimpleTool(806, wx.Bitmap('icons/stock_paste.png'), 'Paste',
        self.toolbar.AddSeparator()

        self.toolbar.AddSimpleTool(807, wx.Bitmap('icons/stock_exit.png'), 'Exit', ''
        self.toolbar.Realize()

        self.Bind(wx.EVT_TOOL, self.NewApplication, id=801)
        self.Bind(wx.EVT_TOOL, self.OnOpenFile, id=802)
        self.Bind(wx.EVT_TOOL, self.OnSaveFile, id=803)
        self.Bind(wx.EVT_TOOL, self.OnCut, id=804)
        self.Bind(wx.EVT_TOOL, self.OnCopy, id=805)
        self.Bind(wx.EVT_TOOL, self.OnPaste, id=806)
        self.Bind(wx.EVT_TOOL, self.QuitApplication, id=807)

        self.text = wx.TextCtrl(self, 1000, '', size=(-1, -1), style=wx.TE_MULTILINE
                | wx.TE_PROCESS_ENTER)
        self.text.SetFocus()
        self.text.Bind(wx.EVT_TEXT, self.OnTextChanged, id=1000)
        self.text.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)

        self.Bind(wx.EVT_CLOSE, self.QuitApplication)

        self.StatusBar()

        self.Centre()
        self.Show(True)

    def NewApplication(self, event):
        editor = Editor(None, -1, 'Editor')
        editor.Centre()
        editor.Show()

    def OnOpenFile(self, event):
        file_name = os.path.basename(self.last_name_saved)
        if self.modify:
            dlg = wx.MessageDialog(self, 'Save changes?', '', wx.YES_NO | wx.YES_DEFA
             wx.CANCEL | wx.ICON_QUESTION)
            val = dlg.ShowModal()
            if val == wx.ID_YES:
                self.OnSaveFile(event)
```

```
            self.DoOpenFile()
        elif val == wx.ID_CANCEL:
            dlg.Destroy()
        else:
            self.DoOpenFile()
    else:
        self.DoOpenFile()

def DoOpenFile(self):
    wcd = 'All files (*)|*|Editor files (*.ef)|*.ef|'
    dir = os.getcwd()
    open_dlg = wx.FileDialog(self, message='Choose a file', defaultDir=dir, defau
                    wildcard=wcd, style=wx.OPEN|wx.CHANGE_DIR)
    if open_dlg.ShowModal() == wx.ID_OK:
        path = open_dlg.GetPath()

        try:
            file = open(path, 'r')
            text = file.read()
            file.close()
            if self.text.GetLastPosition():
                self.text.Clear()
            self.text.WriteText(text)
            self.last_name_saved = path
            self.statusbar.SetStatusText('', 1)
            self.modify = False

        except IOError, error:
            dlg = wx.MessageDialog(self, 'Error opening file\n' + str(error))
            dlg.ShowModal()

        except UnicodeDecodeError, error:
            dlg = wx.MessageDialog(self, 'Error opening file\n' + str(error))
            dlg.ShowModal()

    open_dlg.Destroy()

def OnSaveFile(self, event):
    if self.last_name_saved:

        try:
            file = open(self.last_name_saved, 'w')
            text = self.text.GetValue()
            file.write(text)
            file.close()
            self.statusbar.SetStatusText(os.path.basename(self.last_name_saved) +
            self.modify = False
            self.statusbar.SetStatusText('', 1)

        except IOError, error:
            dlg = wx.MessageDialog(self, 'Error saving file\n' + str(error))
            dlg.ShowModal()
    else:
        self.OnSaveAsFile(event)

def OnSaveAsFile(self, event):
    wcd='All files(*)|*|Editor files (*.ef)|*.ef|'
    dir = os.getcwd()
    save_dlg = wx.FileDialog(self, message='Save file as...', defaultDir=dir, def
                    wildcard=wcd, style=wx.SAVE | wx.OVERWRITE_PROMPT)
    if save_dlg.ShowModal() == wx.ID_OK:
        path = save_dlg.GetPath()

        try:
            file = open(path, 'w')
            text = self.text.GetValue()
            file.write(text)
```

```
            file.close()
            self.last_name_saved = os.path.basename(path)
            self.statusbar.SetStatusText(self.last_name_saved + ' saved', 0)
            self.modify = False
            self.statusbar.SetStatusText('', 1)

        except IOError, error:
            dlg = wx.MessageDialog(self, 'Error saving file\n' + str(error))
            dlg.ShowModal()
    save_dlg.Destroy()

def OnCut(self, event):
    self.text.Cut()

def OnCopy(self, event):
    self.text.Copy()

def OnPaste(self, event):
    self.text.Paste()

def QuitApplication(self, event):
    if self.modify:
        dlg = wx.MessageDialog(self, 'Save before Exit?', '', wx.YES_NO | wx.YES_
                    wx.CANCEL | wx.ICON_QUESTION)
        val = dlg.ShowModal()
        if val == wx.ID_YES:
            self.OnSaveFile(event)
            if not self.modify:
                wx.Exit()
        elif val == wx.ID_CANCEL:
            dlg.Destroy()
        else:
            self.Destroy()
    else:
        self.Destroy()

def OnDelete(self, event):
    frm, to = self.text.GetSelection()
    self.text.Remove(frm, to)

def OnSelectAll(self, event):
    self.text.SelectAll()

def OnTextChanged(self, event):
    self.modify = True
    self.statusbar.SetStatusText(' modified', 1)
    event.Skip()

def OnKeyDown(self, event):
    keycode = event.GetKeyCode()
    if keycode == wx.WXK_INSERT:
        if not self.replace:
            self.statusbar.SetStatusText('INS', 2)
            self.replace = True
        else:
            self.statusbar.SetStatusText('', 2)
            self.replace = False
    event.Skip()

def ToggleStatusBar(self, event):
    if self.statusbar.IsShown():
        self.statusbar.Hide()
    else:
        self.statusbar.Show()

def StatusBar(self):
    self.statusbar = self.CreateStatusBar()
```

```
        self.statusbar.SetFieldsCount(3)
        self.statusbar.SetStatusWidths([-5, -2, -1])

    def OnAbout(self, event):
        dlg = wx.MessageDialog(self, '\tEditor\t\n Another Tutorial\njan bodnar 2005-
                               'About Editor', wx.OK | wx.ICON_INFORMATION)
        dlg.ShowModal()
        dlg.Destroy()

app = wx.App()
Editor(None, -1, 'Editor')
app.MainLoop()
```

Kika

Kika is a gript that connects to an ftp site. If a login is successfull, Kika shows a connected icon on the statusbar. Otherwise, a disconnected icon is displayed. We use an ftplib module from the python standard library. If you do not have an ftp account, you can try to login to some anonymous ftp sites.

Figure: Kika

```
#!/usr/bin/python

# kika.py

from ftplib import FTP, all_errors
import wx

class MyStatusBar(wx.StatusBar):
    def __init__(self, parent):
        wx.StatusBar.__init__(self, parent)

        self.SetFieldsCount(2)
        self.SetStatusText('Welcome to Kika', 0)
        self.SetStatusWidths([-5, -2])
        self.icon = wx.StaticBitmap(self, -1, wx.Bitmap('icons/disconnected.png'))
        self.Bind(wx.EVT_SIZE, self.OnSize)
```

```
        self.PlaceIcon()

    def PlaceIcon(self):
        rect = self.GetFieldRect(1)
        self.icon.SetPosition((rect.x+3, rect.y+3))

    def OnSize(self, event):
        self.PlaceIcon()


class Kika(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(250, 270))

        wx.StaticText(self, -1, 'Ftp site', (10, 20))
        wx.StaticText(self, -1, 'Login', (10, 60))
        wx.StaticText(self, -1, 'Password', (10, 100))

        self.ftpsite = wx.TextCtrl(self, -1, '',  (110, 15), (120, -1))
        self.login = wx.TextCtrl(self, -1, '',  (110, 55), (120, -1))
        self.password = wx.TextCtrl(self, -1, '',  (110, 95), (120, -1), style=wx.TE_

        self.ftp = None

        con = wx.Button(self, 1, 'Connect', (10, 160))
        discon = wx.Button(self, 2, 'DisConnect', (120, 160))

        self.Bind(wx.EVT_BUTTON, self.OnConnect, id=1)
        self.Bind(wx.EVT_BUTTON, self.OnDisConnect, id=2)

        self.statusbar = MyStatusBar(self)
        self.SetStatusBar(self.statusbar)
        self.Centre()
        self.Show()

    def OnConnect(self, event):
        if not self.ftp:
            ftpsite = self.ftpsite.GetValue()
            login = self.login.GetValue()
            password = self.password.GetValue()

            try:
                self.ftp = FTP(ftpsite)
                var = self.ftp.login(login, password)
                self.statusbar.SetStatusText('User connected')
                self.statusbar.icon.SetBitmap(wx.Bitmap('icons/connected.png'))

            except AttributeError:
                self.statusbar.SetForegroundColour(wx.RED)
                self.statusbar.SetStatusText('Incorrect params')
                self.ftp = None

            except all_errors, err:
                self.statusbar.SetStatusText(str(err))
                self.ftp = None

    def OnDisConnect(self, event):
        if self.ftp:
            self.ftp.quit()
            self.ftp = None
            self.statusbar.SetStatusText('User disconnected')
            self.statusbar.icon.SetBitmap(wx.Bitmap('icons/disconnected.png'))


app = wx.App()
Kika(None, -1, 'Kika')
app.MainLoop()
```

Notice that each time the window is resized, we must position our icon to a new place.

```
def PlaceIcon(self):
    rect = self.GetFieldRect(1)
    self.icon.SetPosition((rect.x+3, rect.y+3))
```

Puzzle

In this gript, we introduce a puzzle game. We have an image of a Sid character from the Ice Age movie. It is cut into 9 pieces and shuffled. The goal is to form the picture.

Figure: Puzzle

```
#!/usr/bin/python

# puzzle.py

import wx
import random

class Puzzle(wx.Dialog):
    def __init__(self, parent, id, title):
        wx.Dialog.__init__(self, parent, id, title)

        images = ['images/one.jpg', 'images/two.jpg', 'images/three.jpg', 'images/fou
                'images/five.jpg', 'images/six.jpg', 'images/seven.jpg', 'images/eigh

        self.pos = [ [0, 1, 2], [3, 4, 5], [6, 7, 8] ]

        self.sizer = wx.GridSizer(3, 3, 0, 0)

        numbers = [0, 1, 2, 3, 4, 5, 6, 7]
        random.shuffle(numbers)

        for i in numbers:
                button = wx.BitmapButton(self, i, wx.Bitmap(images[i]))
                button.Bind(wx.EVT_BUTTON, self.OnPressButton, id=button.GetId())
```

```
                self.sizer.Add(button)

        self.panel = wx.Button(self, -1, size=(112, 82))
        self.sizer.Add(self.panel)

        self.SetSizerAndFit(self.sizer)
        self.Centre()
        self.ShowModal()
        self.Destroy()

    def OnPressButton(self, event):

        button = event.GetEventObject()
        sizeX = self.panel.GetSize().x
        sizeY = self.panel.GetSize().y

        buttonX = button.GetPosition().x
        buttonY = button.GetPosition().y
        panelX = self.panel.GetPosition().x
        panelY = self.panel.GetPosition().y
        buttonPosX = buttonX / sizeX
        buttonPosY = buttonY / sizeY

        buttonIndex = self.pos[buttonPosY][buttonPosX]
        if (buttonX == panelX) and (panelY - buttonY) == sizeY:
            self.sizer.Remove(self.panel)
            self.sizer.Remove(button)
            self.sizer.Insert(buttonIndex, self.panel)
            self.sizer.Insert(buttonIndex+3, button)
            self.sizer.Layout()

        if (buttonX == panelX) and (panelY - buttonY) == -sizeY:
            self.sizer.Remove(self.panel)
            self.sizer.Remove(button)
            self.sizer.Insert(buttonIndex-3, button)
            self.sizer.Insert(buttonIndex, self.panel)
            self.sizer.Layout()

        if (buttonY == panelY) and (panelX - buttonX) == sizeX:
            self.sizer.Remove(self.panel)
            self.sizer.Remove(button)
            self.sizer.Insert(buttonIndex, self.panel)
            self.sizer.Insert(buttonIndex+1, button)
            self.sizer.Layout()

        if (buttonY == panelY) and (panelX - buttonX) == -sizeX:
            self.sizer.Remove(self.panel)
            self.sizer.Remove(button)
            self.sizer.Insert(buttonIndex-1, button)
            self.sizer.Insert(buttonIndex, self.panel)
            self.sizer.Layout()

app = wx.App()
Puzzle(None, -1, 'Puzzle')
app.MainLoop()
```

```
images = ['images/one.jpg', 'images/two.jpg', 'images/three.jpg', 'images/four.jpg',
        'images/five.jpg', 'images/six.jpg', 'images/seven.jpg', 'images/eight.jpg']
```

The picture was cut into 9 parts of 100x70 size. I did it with the Gimp program. Each part of the picture is placed on one button widget. Except one.

```
self.sizer = wx.GridSizer(3, 3, 0, 0)
```

For this gript, *wx.GridSizer* fits ideally.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7]
random.shuffle(numbers)
```

We have eight numbers. Those numbers are shuffled so that we have a random number order. Each time we start the gript, we will have a different order of bitmaps.

```
self.panel = wx.Button(self, -1, size=(112, 82))
self.sizer.Add(self.panel)
```

This button has no bitmap. It is the 'travelling' button. It always exchanges it's position with the hitted button.

Home ‡ Contents ‡ Top of Page

The Tetris game in wxPython

Tetris

The tetris game is one of the most popular computer games ever created. The original game was designed and programmed by a russian programmer **Alexey Pajitnov** in 1985. Since then, tetris is available on almost every computer platform in lots of variations. Even my mobile phone has a modified version of the tetris game.

Tetris is called a falling block puzzle game. In this game, we have seven different shapes called **tetrominoes**. S-shape, Z-shape, T-shape, L-shape, Line-shape, MirroredL-shape and a Square-shape. Each of these shapes is formed with four squares. The shapes are falling down the board. The object of the tetris game is to move and rotate the shapes, so that they fit as much as possible. If we manage to form a row, the row is destroyed and we score. We play the tetris game until we top out.

Figure: Tetrominoes

wxPython is a toolkit designed to create applications. There are other libraries which are targeted at creating computer games. Nevertheless, wxPython and other application toolkits can be used to create games.

The development

We do not have images for our tetris game, we draw the tetrominoes using the drawing API available in the wxPython programming toolkit. Behind every computer game, there is a mathematical model. So it is in tetris.

Some ideas behind the game.

- We use **wx.Timer** to create a game cycle

- The tetrominoes are drawn
- The shapes move on a square by square basis (not pixel by pixel)
- Mathematically a board is a simple list of numbers

The following example is a modified version of the tetris game, available with PyQt4 installation files.

```
#!/usr/bin/python

# tetris.py

import wx
import random

class Tetris(wx.Frame):
    def __init__(self, parent, id, title):
        wx.Frame.__init__(self, parent, id, title, size=(180, 380))

        self.statusbar = self.CreateStatusBar()
        self.statusbar.SetStatusText('0')
        self.board = Board(self)
        self.board.SetFocus()
        self.board.start()

        self.Centre()
        self.Show(True)


class Board(wx.Panel):
    BoardWidth = 10
    BoardHeight = 22
    Speed = 300
    ID_TIMER = 1

    def __init__(self, parent):
        wx.Panel.__init__(self, parent)

        self.timer = wx.Timer(self, Board.ID_TIMER)
        self.isWaitingAfterLine = False
        self.curPiece = Shape()
        self.nextPiece = Shape()
        self.curX = 0
        self.curY = 0
        self.numLinesRemoved = 0
        self.board = []

        self.isStarted = False
        self.isPaused = False

        self.Bind(wx.EVT_PAINT, self.OnPaint)
        self.Bind(wx.EVT_KEY_DOWN, self.OnKeyDown)
        self.Bind(wx.EVT_TIMER, self.OnTimer, id=Board.ID_TIMER)

        self.clearBoard()

    def shapeAt(self, x, y):
```

```
        return self.board[(y * Board.BoardWidth) + x]

    def setShapeAt(self, x, y, shape):
        self.board[(y * Board.BoardWidth) + x] = shape

    def squareWidth(self):
        return self.GetClientSize().GetWidth() / Board.BoardWidth

    def squareHeight(self):
        return self.GetClientSize().GetHeight() / Board.BoardHeight

    def start(self):
        if self.isPaused:
            return

        self.isStarted = True
        self.isWaitingAfterLine = False
        self.numLinesRemoved = 0
        self.clearBoard()

        self.newPiece()
        self.timer.Start(Board.Speed)

    def pause(self):
        if not self.isStarted:
            return

        self.isPaused = not self.isPaused
        statusbar = self.GetParent().statusbar

        if self.isPaused:
            self.timer.Stop()
            statusbar.SetStatusText('paused')
        else:
            self.timer.Start(Board.Speed)
            statusbar.SetStatusText(str(self.numLinesRemoved))

        self.Refresh()

    def clearBoard(self):
        for i in range(Board.BoardHeight * Board.BoardWidth):
            self.board.append(Tetrominoes.NoShape)

    def OnPaint(self, event):

        dc = wx.PaintDC(self)

        size = self.GetClientSize()
        boardTop = size.GetHeight() - Board.BoardHeight * self.squareHeight(

        for i in range(Board.BoardHeight):
            for j in range(Board.BoardWidth):
                shape = self.shapeAt(j, Board.BoardHeight - i - 1)
                if shape != Tetrominoes.NoShape:
                    self.drawSquare(dc,
                        0 + j * self.squareWidth(),
                        boardTop + i * self.squareHeight(), shape)

        if self.curPiece.shape() != Tetrominoes.NoShape:
            for i in range(4):
                x = self.curX + self.curPiece.x(i)
                y = self.curY - self.curPiece.y(i)
```

```
            self.drawSquare(dc, 0 + x * self.squareWidth(),
                boardTop + (Board.BoardHeight - y - 1) * self.squareHeigl
                self.curPiece.shape())


    def OnKeyDown(self, event):
        if not self.isStarted or self.curPiece.shape() == Tetrominoes.NoShape
            event.Skip()
            return

        keycode = event.GetKeyCode()

        if keycode == ord('P') or keycode == ord('p'):
            self.pause()
            return
        if self.isPaused:
            return
        elif keycode == wx.WXK_LEFT:
            self.tryMove(self.curPiece, self.curX - 1, self.curY)
        elif keycode == wx.WXK_RIGHT:
            self.tryMove(self.curPiece, self.curX + 1, self.curY)
        elif keycode == wx.WXK_DOWN:
            self.tryMove(self.curPiece.rotatedRight(), self.curX, self.curY)
        elif keycode == wx.WXK_UP:
            self.tryMove(self.curPiece.rotatedLeft(), self.curX, self.curY)
        elif keycode == wx.WXK_SPACE:
            self.dropDown()
        elif keycode == ord('D') or keycode == ord('d'):
            self.oneLineDown()
        else:
            event.Skip()


    def OnTimer(self, event):
        if event.GetId() == Board.ID_TIMER:
            if self.isWaitingAfterLine:
                self.isWaitingAfterLine = False
                self.newPiece()
            else:
                self.oneLineDown()
        else:
            event.Skip()


    def dropDown(self):
        newY = self.curY
        while newY > 0:
            if not self.tryMove(self.curPiece, self.curX, newY - 1):
                break
            newY -= 1

        self.pieceDropped()

    def oneLineDown(self):
        if not self.tryMove(self.curPiece, self.curX, self.curY - 1):
            self.pieceDropped()


    def pieceDropped(self):
        for i in range(4):
            x = self.curX + self.curPiece.x(i)
            y = self.curY - self.curPiece.y(i)
```

```
            self.setShapeAt(x, y, self.curPiece.shape())

        self.removeFullLines()

        if not self.isWaitingAfterLine:
            self.newPiece()


    def removeFullLines(self):
        numFullLines = 0

        statusbar = self.GetParent().statusbar

        rowsToRemove = []

        for i in range(Board.BoardHeight):
            n = 0
            for j in range(Board.BoardWidth):
                if not self.shapeAt(j, i) == Tetrominoes.NoShape:
                    n = n + 1

            if n == 10:
                rowsToRemove.append(i)

        rowsToRemove.reverse()

        for m in rowsToRemove:
            for k in range(m, Board.BoardHeight):
                for l in range(Board.BoardWidth):
                    self.setShapeAt(l, k, self.shapeAt(l, k + 1))

        numFullLines = numFullLines + len(rowsToRemove)

        if numFullLines > 0:
            self.numLinesRemoved = self.numLinesRemoved + numFullLines
            statusbar.SetStatusText(str(self.numLinesRemoved))
            self.isWaitingAfterLine = True
            self.curPiece.setShape(Tetrominoes.NoShape)
            self.Refresh()


    def newPiece(self):
        self.curPiece = self.nextPiece
        statusbar = self.GetParent().statusbar
        self.nextPiece.setRandomShape()
        self.curX = Board.BoardWidth / 2 + 1
        self.curY = Board.BoardHeight - 1 + self.curPiece.minY()

        if not self.tryMove(self.curPiece, self.curX, self.curY):
            self.curPiece.setShape(Tetrominoes.NoShape)
            self.timer.Stop()
            self.isStarted = False
            statusbar.SetStatusText('Game over')

    def tryMove(self, newPiece, newX, newY):
        for i in range(4):
            x = newX + newPiece.x(i)
            y = newY - newPiece.y(i)
            if x < 0 or x >= Board.BoardWidth or y < 0 or y >= Board.BoardHe
                return False
            if self.shapeAt(x, y) != Tetrominoes.NoShape:
                return False
```

```
        self.curPiece = newPiece
        self.curX = newX
        self.curY = newY
        self.Refresh()
        return True


    def drawSquare(self, dc, x, y, shape):
        colors = ['#000000', '#CC6666', '#66CC66', '#6666CC',
                  '#CCCC66', '#CC66CC', '#66CCCC', '#DAAA00']

        light = ['#000000', '#F89FAB', '#79FC79', '#7979FC',
                 '#FCFC79', '#FC79FC', '#79FCFC', '#FCC600']

        dark = ['#000000', '#803C3B', '#3B803B', '#3B3B80',
                 '#80803B', '#803B80', '#3B8080', '#806200']

        pen = wx.Pen(light[shape])
        pen.SetCap(wx.CAP_PROJECTING)
        dc.SetPen(pen)

        dc.DrawLine(x, y + self.squareHeight() - 1, x, y)
        dc.DrawLine(x, y, x + self.squareWidth() - 1, y)

        darkpen = wx.Pen(dark[shape])
        darkpen.SetCap(wx.CAP_PROJECTING)
        dc.SetPen(darkpen)

        dc.DrawLine(x + 1, y + self.squareHeight() - 1,
            x + self.squareWidth() - 1, y + self.squareHeight() - 1)
        dc.DrawLine(x + self.squareWidth() - 1,
            y + self.squareHeight() - 1, x + self.squareWidth() - 1, y + 1)

        dc.SetPen(wx.TRANSPARENT_PEN)
        dc.SetBrush(wx.Brush(colors[shape]))
        dc.DrawRectangle(x + 1, y + 1, self.squareWidth() - 2,
            self.squareHeight() - 2)


class Tetrominoes(object):
    NoShape = 0
    ZShape = 1
    SShape = 2
    LineShape = 3
    TShape = 4
    SquareShape = 5
    LShape = 6
    MirroredLShape = 7


class Shape(object):
    coordsTable = (
        ((0, 0),     (0, 0),     (0, 0),     (0, 0)),
        ((0, -1),    (0, 0),     (-1, 0),    (-1, 1)),
        ((0, -1),    (0, 0),     (1, 0),     (1, 1)),
        ((0, -1),    (0, 0),     (0, 1),     (0, 2)),
        ((-1, 0),    (0, 0),     (1, 0),     (0, 1)),
        ((0, 0),     (1, 0),     (0, 1),     (1, 1)),
        ((-1, -1),   (0, -1),    (0, 0),     (0, 1)),
        ((1, -1),    (0, -1),    (0, 0),     (0, 1))
    )
```

```
def __init__(self):
    self.coords = [[0,0] for i in range(4)]
    self.pieceShape = Tetrominoes.NoShape

    self.setShape(Tetrominoes.NoShape)

def shape(self):
    return self.pieceShape

def setShape(self, shape):
    table = Shape.coordsTable[shape]
    for i in range(4):
        for j in range(2):
            self.coords[i][j] = table[i][j]

    self.pieceShape = shape

def setRandomShape(self):
    self.setShape(random.randint(1, 7))

def x(self, index):
    return self.coords[index][0]

def y(self, index):
    return self.coords[index][1]

def setX(self, index, x):
    self.coords[index][0] = x

def setY(self, index, y):
    self.coords[index][1] = y

def minX(self):
    m = self.coords[0][0]
    for i in range(4):
        m = min(m, self.coords[i][0])

    return m

def maxX(self):
    m = self.coords[0][0]
    for i in range(4):
        m = max(m, self.coords[i][0])

    return m

def minY(self):
    m = self.coords[0][1]
    for i in range(4):
        m = min(m, self.coords[i][1])

    return m

def maxY(self):
    m = self.coords[0][1]
    for i in range(4):
        m = max(m, self.coords[i][1])

    return m

def rotatedLeft(self):
```

```
        if self.pieceShape == Tetrominoes.SquareShape:
            return self

        result = Shape()
        result.pieceShape = self.pieceShape
        for i in range(4):
            result.setX(i, self.y(i))
            result.setY(i, -self.x(i))

        return result

    def rotatedRight(self):
        if self.pieceShape == Tetrominoes.SquareShape:
            return self

        result = Shape()
        result.pieceShape = self.pieceShape
        for i in range(4):
            result.setX(i, -self.y(i))
            result.setY(i, self.x(i))

        return result


app = wx.App()
Tetris(None, -1, 'Tetris')
app.MainLoop()
```

I have simplified the game a bit, so that it is easier to understand. The game starts immediately, after it is launched. We can pause the game by pressing the p key. The space key will drop the tetris piece immediately to the bottom. The d key will drop the piece one line down. (It can be used to speed up the falling a bit.) The game goes at constant speed, no acceleration is implemented. The score is the number of lines, that we have removed.

```
...
self.curX = 0
self.curY = 0
self.numLinesRemoved = 0
self.board = []
...
```

Before we start the game cycle, we initialize some important variables. The **self.board** variable is a list of numbers from 0 ... 7. It represents the position of various shapes and remains of the shapes on the board.

```
for i in range(Board.BoardHeight):
    for j in range(Board.BoardWidth):
        shape = self.shapeAt(j, Board.BoardHeight - i - 1)
        if shape != Tetrominoes.NoShape:
            self.drawSquare(dc,
```

```
                0 + j * self.squareWidth(),
                boardTop + i * self.squareHeight(), shape)
```

The painting of the game is divided into two steps. In the first step, we draw all the shapes, or remains of the shapes, that have been dropped to the bottom of the board. All the squares are rememberd in the **self.board** list variable. We access it using the shapeAt() method.

```
if self.curPiece.shape() != Tetrominoes.NoShape:
    for i in range(4):
        x = self.curX + self.curPiece.x(i)
        y = self.curY - self.curPiece.y(i)
        self.drawSquare(dc, 0 + x * self.squareWidth(),
            boardTop + (Board.BoardHeight - y - 1) * self.squareHeight(),
            self.curPiece.shape())
```

The next step is drawing of the actual piece, that is falling down.

```
elif keycode == wx.WXK_LEFT:
    self.tryMove(self.curPiece, self.curX - 1, self.curY)
```

In the **OnKeyDown()** method we check for pressed keys. If we press the left arrow key, we try to move the piece to the left. We say try, because the piece might not be able to move.

```
def tryMove(self, newPiece, newX, newY):
    for i in range(4):
        x = newX + newPiece.x(i)
        y = newY - newPiece.y(i)
        if x < 0 or x >= Board.BoardWidth or y < 0 or y >= Board.BoardHeight:
            return False
        if self.shapeAt(x, y) != Tetrominoes.NoShape:
            return False

    self.curPiece = newPiece
    self.curX = newX
    self.curY = newY
    self.Refresh()
    return True
```

In the **tryMove()** method we try to move our shapes. If the shape is at the edge of the board or is adjacent to some other piece, we return false. Otherwise we place the current falling piece to a new position and return true.

```
def OnTimer(self, event):
    if event.GetId() == Board.ID_TIMER:
        if self.isWaitingAfterLine:
            self.isWaitingAfterLine = False
            self.newPiece()
        else:
            self.oneLineDown()
    else:
        event.Skip()
```

In the **OnTimer()** method we either create a new piece, after the previous one was dropped to the bottom, or we move a falling piece one line down.

```
def removeFullLines(self):
    numFullLines = 0

    rowsToRemove = []

    for i in range(Board.BoardHeight):
        n = 0
        for j in range(Board.BoardWidth):
            if not self.shapeAt(j, i) == Tetrominoes.NoShape:
                n = n + 1

        if n == 10:
            rowsToRemove.append(i)

     rowsToRemove.reverse()

     for m in rowsToRemove:
         for k in range(m, Board.BoardHeight):
             for l in range(Board.BoardWidth):
                 self.setShapeAt(l, k, self.shapeAt(l, k + 1))
...
```

If the piece hits the bottom, we call the **removeFullLines()** method. First we find out all full lines. And we remove them. We do it by moving all lines above the current full line to be removed one line down. Notice, that we reverse the order of the lines to be removed. Otherwise, it would not work correctly. In our case we use a **naive gravity**. This means, that the pieces may be floating above empty gaps.

```
def newPiece(self):
    self.curPiece = self.nextPiece
    statusbar = self.GetParent().statusbar
    self.nextPiece.setRandomShape()
    self.curX = Board.BoardWidth / 2 + 1
    self.curY = Board.BoardHeight - 1 + self.curPiece.minY()

    if not self.tryMove(self.curPiece, self.curX, self.curY):
        self.curPiece.setShape(Tetrominoes.NoShape)
        self.timer.Stop()
        self.isStarted = False
        statusbar.SetStatusText('Game over')
```

The **newPiece()** method creates randomly a new tetris piece. If the piece cannot go into it's initial position, the game is over.

The **Shape** class saves information about the tetris piece.

```
self.coords = [[0,0] for i in range(4)]
```

Upon creation we create an empty coordinates list. The list will save the coordinates of the tetris piece. For example, these tuples (0, -1), (0, 0), (1, 0), (1, 1) represent a rotated S-shape. The following diagram illustrates the shape.

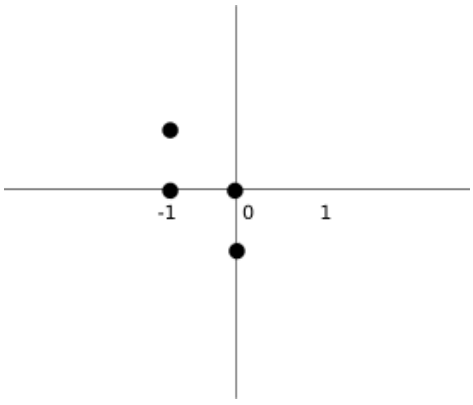


Figure: Coordinates

When we draw the current falling piece, we draw it at **self.curX**, **self.curY position**. Then we look at the coordinates table and draw all the four squares.

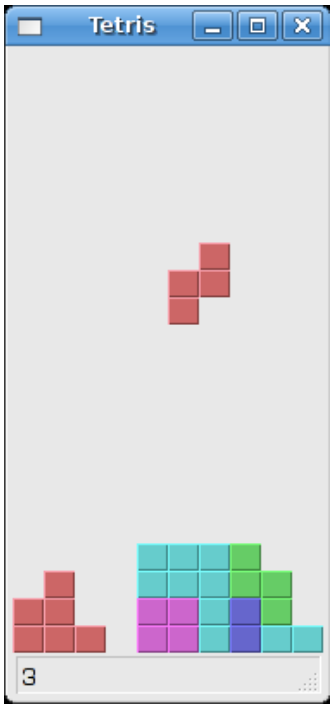


Figure: Tetris

Home ǂ Contents ǂ Top of Page