Python and GTK+

The traditional way of doing things in UNIX® is with the command line. Want a directory listing? Run "`ls`". Want to pause the listing after each screenful? Run "`ls | more`". Each command has a well-defined task to perform, it has an input and an output, and commands can be chained together. It is a very powerful idea, and one that Red Hat Linux builds on too. However, who wants to type things on the keyboard all the time and memorise all the options that different commands use? Today, most people prefer to be able to point and click with their mouse or touchpad.

Graphical applications—with windows and buttons and so on—allow people to point and click. In the Red Hat Linux world we have a windowing system called, simply, X. It provides simple drawing services for programs to use, such as drawing a line, or a box, or a filled region. This level of control is a bit too fine-grained to appeal to the average programmer though, and toolkits are available to make the task easier. Some examples are Qt (which KDE uses), GTK+ (which GNOME uses), and Tcl/Tk. They provide routines for drawing buttons, text boxes, radio buttons, and so on. All these graphical elements are collectively called *widgets*.

Drawing buttons on the screen is not the only task for an X toolkit. It also has to provide methods for navigating the graphical interface using just the keyboard, for adjusting the size and contrast of the widgets, and other things that make the interface more usable in general and, most importantly, accessible. By this I mean usable by people who are visually impaired or have problems using a mouse. For example, you should be able to use the TAB and arrows keys to move the focus around the widgets.

Each toolkit has its preferred programming language, often the one it was written in: Tk applications are generally written in Tcl; Qt applications are generally C++; GTK+ applications are often C. It is possible to use toolkits with other programming languages so long as there are *bindings* (linguistic glue) to map functions in the toolkit library to the new language. One good example is in the Python bindings for GTK+. We'll concentrate on them in this article.

Python as a programming language

Python, if you have not come across it yet, is a programming language which is quite easy to read. It is thin on rules, and can be picked up very quickly. It does not need to be compiled, but instead is interpreted by the Python program (the *interpreter*). This makes it quick and easy to make small changes and test them out. On the negative side, it also means that some types of error which a compiled language would find at compile time will be harder to find.

Like most other modern programming languages, Python has the concept of software objects with associated methods and variables—in other words it is an object-oriented language.

There is a shallower learning curve than with C because Python programs have a more straight-forward look about them. You don't need to remember to put semi-colons all over the place as in C, and the way Python uses indentation to infer scope is intuitive. For example, when you want a piece of code to execute only under some particular circumstance, you will write something like this:

```
if particular_condition:
       do_things ()
       do_more_things ()

print "Done!"
```

It looks just like pseudo-code. The fact that the two lines following the **if** line are indented tells Python that they are both to be executed if the condition is true, and both to be skipped if it is not. The program would print "`Done!`" in either case. In C this scoping is marked by curly braces.

You do not need to declare variables before you use them. Assigning a value to a variable (like "`x=3`") is enough to make it appear. As well as numbers and strings, Python understands how lists work, as well as associative arrays. (An associative array is like a dictionary, in that you use a key to look up a value.)

You can iterate over a list using "**for** `var` **in** `list:`", and you can create a list of numbers starting at zero using the **range** function. So for instance, here is how you can count to ten:

```
for i in range (10):
       print i + 1
```

Python looks similar to Visual Basic, but in a good way: while having quite a simple structure it is deceptively powerful. As with Visual Basic, it is extensible and can be used to control more complicated program modules. The point about this article of course is that Python can control GTK+ widgets.

```
greeting = "hello world".capitalize ()
for i in range (1, 3):
  # This will display things like 'hello world 1'
  print "%s %d" % (greeting, i)

  if i == 2:
    print "And one for luck!"
```

Illustration 1 Simple Python example

Illustration 1 is a snippet of Python showing some more features of the language. The string "`hello world`" is used as an object, and its **capitalize** method is invoked. Most things in Python can be treated in this sort of way. You can create your own objects and methods too of course. To run the example program, create a file called `hello.py` with the contents as shown, and use `python ./hello.py` at the command line to execute it.

The **print** line shows how Python can perform string substitution in a similar way to the **printf** function of C. Comments are started with the “#” character and extend to the end of the line, just like in shell scripts.

When Python encounters an error in the program it is interpreting, it will give you a *traceback*, like this:

```
Traceback (most recent call last):
  File "./cross.py", line 34, in ?
    start()
  File "./cross.py", line 32, in start
    help()
  File "./cross.py", line 8, in help
    print "usage: %s" % name
NameError: global name 'name' is not defined
```

This shows the context of the error. As the first line says, the most recent call is at the end. In this case, line 34 of my file has called a function named **start**, and that function has, at line 32, called another function named **help**. The definition of that function tries to use a variable that does not exist. The error messages that Python gives are usually quite explanatory.

You can use Python interactively or give it a script to run. Generally you will only use Python interactively when trying things out. A finished Python program will be a script in a file. To run Python interactively just type `python` at the command line, and you will see something like this:

```
Python 2.2.3 (#1, Oct 15 2003, 23:33:35)
[GCC 3.3.1 20030930 (Red Hat Linux 3.3.1-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This is sometimes called a Python *shell*, since it is a thin wrapper around the Python interpreter itself. To get out of the Python shell and back to the command line press Control+D.

When you run Python interactively, by just typing `python` at the command line, you can use the **dir** function to find out what methods a given object has. For instance, typing `print dir ("hello")` at the Python prompt will list all the methods that string objects provide, including **capitalize**. You can even find out what they do by typing, for example, `help ("hello".capitalize)`. If you do that, you will see the following:

```
Help on built-in function capitalize:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
```

I usually only run Python interactively when I want to remember which methods and functions are available. Functions can be collected into a special kind of object named a *module*, and modules are loaded using **import**. You can use **dir** to find out what functions and variables a module provides; for instance you can load the “`os`” module using “`import os`” and find out what's in it using "`dir(os)`".

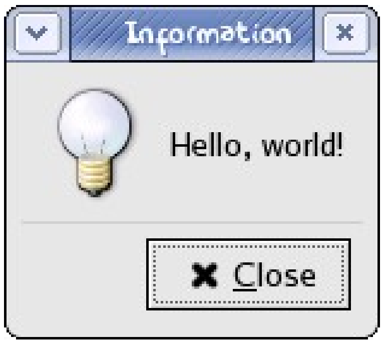
There are some excellent tutorials available for Python, and there are plenty of useful resources on the Python website, www.python.org. The Emacs text editor has keyword highlighting support for Python, and I find its automatic indentation particularly useful.

Python and widgets

Illustration 2 shows a small Python program which displays a window with a button.

```
# Load the GTK+ module
import gtk

# Create the dialog widget
d = gtk.MessageDialog (None, 0,
                       gtk.MESSAGE_INFO,
                       gtk.BUTTONS_CLOSE,
                       "Hello, world!")
# Make it run
d.run ()
```



Hello, world!

Illustration 2 Python widget example

The **MessageDialog** function creates a GTK+ dialog widget. In the example, the variable *d* is the Python object for that widget. Calling its **run** method makes the button clickable. The run method will not return until the dialog is closed. With GTK+ the logic driving the buttons and other widgets takes place in the *main loop*. This loop sits waiting for the user to do something, and then responds. If something happens for which the main program has registered a *callback*, control is passed to it.

The Python bindings for GTK+, called PyGTK, are documented at http://www.gnome.org/~james/pygtk-docs/. Sometimes, however, the GTK+ documentation itself will be more useful—you can find that at http://developer.gnome.org/doc/API/2.0/gtk/. On Red Hat Linux you can find this documentation in the gtk2-devel package; once you have installed it, point your browser at file:///usr/share/gtk-doc/html/gtk/index.html.

Creating dialogs with Glade

You can create GTK+ dialog windows and interfaces using a simple tool called Glade. Using Glade you can slot different types of widgets together to make any sort of window you like, and give each widget a name. In your Python program you can then control the widgets by name.

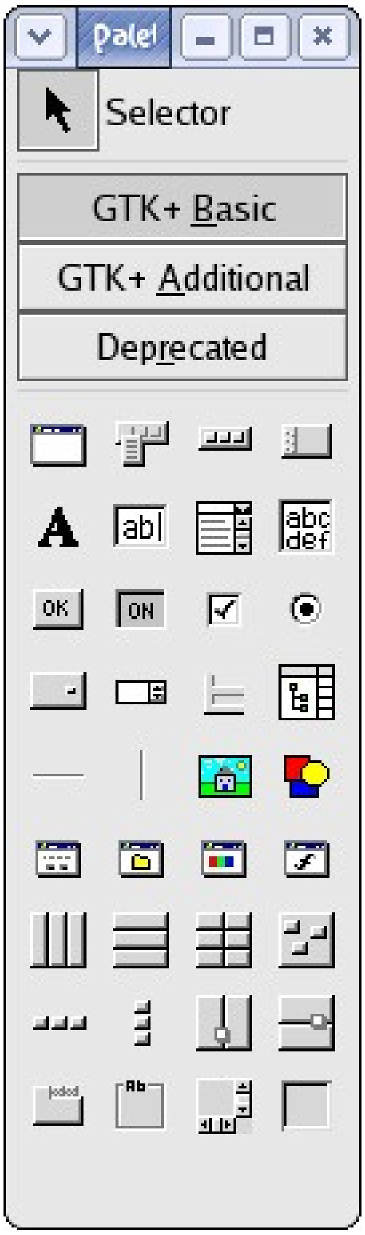


Illustration 3 Glade

You can start Glade by clicking on the **`Red Hat`** menu, then **`Programming`**, then **`Glade Interface Designer`**. Three windows will appear: the main window (with a **`New`** button), the palette, and the properties window. After you click **`New`** on the main window and select to create a new GTK+ project, you are ready to start designing an interface. Illustration 3 shows the palette window, offering a selection of widgets.

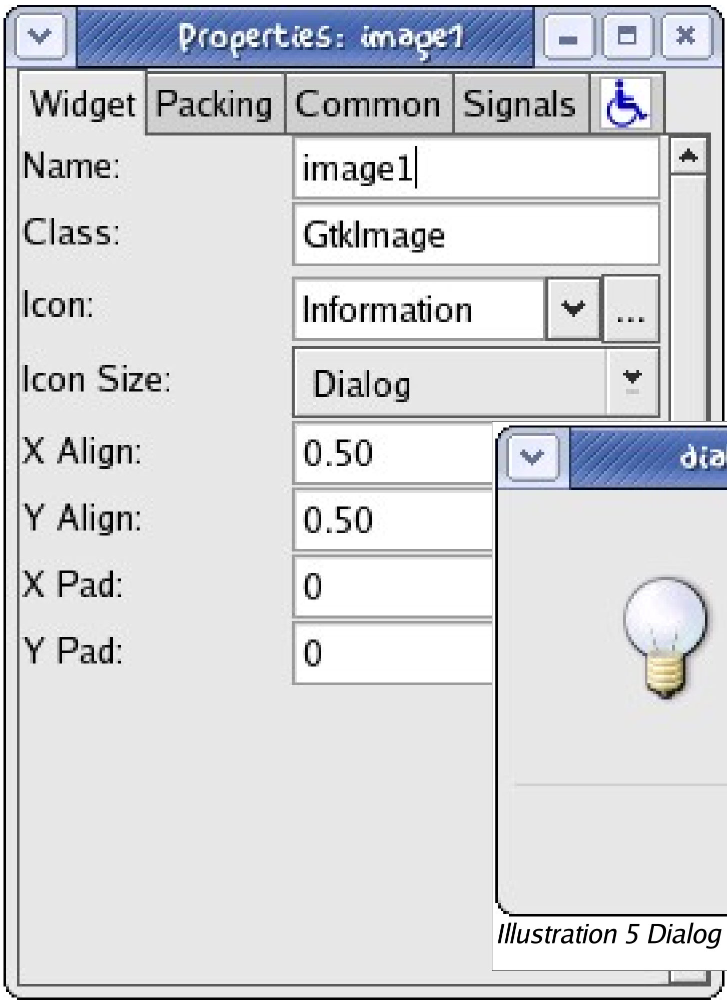
The first widget on the 6th row in the palette is for a dialog window (hover the mouse over it and it tells you "`Dialog`"). Click that, and you will be asked which buttons the dialog should have. Choose **`Close`**, for example, and click **`OK`**. Our dialog window will appear, with a **`Close`** button, and some blank space above it. You can put a widget in that blank space by selecting it from the palette and clicking in the blank space on your dialog window.

Let's carry on and make a dialog that looks like the one in Illustration 2. For that we need a picture of a lightbulb, and some text beside it. The blank space in our dialog window is for a single widget, but we can fit two in there by using a *horizontal box* as our widget. This is a widget container, and just serves to split up the space into parts so that we can put more than one widget in the space. The horizontal box widget is the first on the 7th row of the palette. Select it, and then click in the space on the dialog window. You will be asked how many columns you require; for just the picture and the text, we want 2 columns. A vertical line will appear, dividing the space horizontally in the middle.

Now for the lightbulb picture. Click on the drawing of a house (the tooltip says "`Image`") on the 5th row of the palette, and then on the blank space on the left of the dialog window; the house picture will appear there.

One of the windows that should have appeared when you started the Glade Interface Designer is for editing widget properties. (If not, select **`View`** then **`Show Property Editor`** from the main window, and click again on the house picture in our dialog window.)

The properties of the image widget are displayed, and you can select which picture the widget will display. In the **`Icon`** field, click on the arrow to see the list of "stock" icons (the icons that are provided by GTK+). The lightbulb picture is the **`Information`** icon. Change the icon size to **`Dialog`**. The properties window should then look like Illustration 4, and our dialog window should look something like Illustration 5.



Name: image1
Class: GtkImage
Icon: Information
Icon Size: Dialog
X Align: 0.50
Y Align: 0.50
X Pad: 0
Y Pad: 0

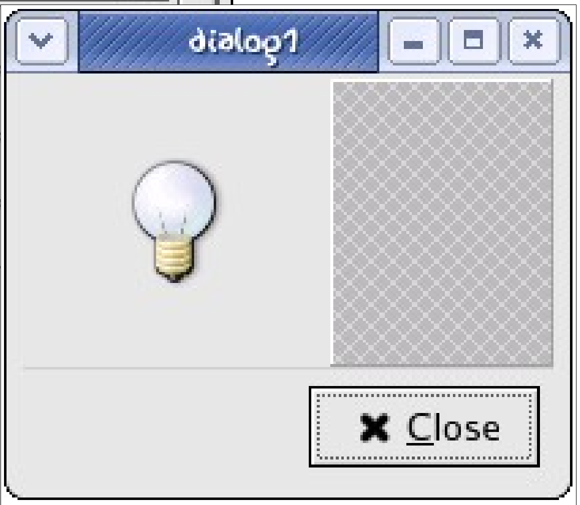


Illustration 5 Dialog window during construction

Illustration 4 Properties window

For the text beside it we need to use a label widget, the first on the 2nd row of the palette. Select it, then click on the remaining blank (actually cross-hatched) space on the right of our dialog window. The text `label1` will appear there, and the properties window will reflect the label's properties. Change the label text to read `Hello, world!`, and we're almost done.

The last thing to change to make the dialog look how we want it to is on the **`Packing`** tab of the properties window: change the **`Expand`** field to be `Yes`. This makes the widget compete for space as much as the image widget does, and so makes the dialog look nicer.

To finish off the example, here is a Python program to do the equivalent of the one in Illustration 2, using Glade. Save your Glade project as `project1.glade` (just press the **`Save`** button and then **`OK`**), and create the

Python program in the same directory:

```
# Load the Glade module
import gtk.glade

# Load the dialog we designed
xml = gtk.glade.XML ("project1.glade")
d = xml.get_widget ("dialog1")

# Make it run
d.run ()
```

The `project1.glade` file contains a description of all the dialogs and windows that we created using Glade; in this case just the `Hello, world!` dialog box. Once it is loaded using **gtk.glade.XML** you can get at the individual widgets (the image widget, the label, the close button, the dialog box itself, and so on) using the **get_widget** function.

Why go to the bother of designing a dialog using Glade at all? Didn't the **MessageDialog** function in Illustration 2 do what we wanted? Well, yes it did, but when the interface starts to get more complicated than a picture and a button you will find that using Glade to create the widgets and connect them together is easier than doing it all inside your Python program.

A small program using Python and Glade

Now we know how to design interfaces and link them up with Python, we can make a simple application. For this example, we will make a printing dialog application. Its task will be to ask the user which print queue to use, and then to send its input to that queue. We'll call it "print", and it can be used like: `print < file.ps`.

This program is actually useful too. Currently the Mozilla web browser is not very good at offering a selection of the available print queues when you print a page—instead you are made to edit an **lpr** command line. Once our program is finished we'll be able to tell Mozilla to use it as its print command, and finally get our drop-down box choosing a queue.

It could also be used as a simple replacement for `lpr` in other programs. This is the sort of job that `kprinter` does already, but we will aim to just make a really simple tool, for the sake of demonstration.

For finding out which queues are available, we will cheat and assume that the CUPS spooler is going to be in use. In this case there is a rather easy way of finding the queues. Starting with Red Hat Linux 9, the Red Hat printer configuration tool provides a Python module for discovering queues. We will use that.

Beginning with the Glade part, we want to make a dialog window with two buttons, **`Cancel`** and **`Print`**. That combination is not one of the preset choices, so instead choose the closest match (**`Cancel, OK`**) which we can then change later. When the dialog window has been created, change its title from `dialog1` to `Print` using the **`Properties`** window. Distinct from the window title, you need to set the widget name of the dialog as well, since we will use it in our Python code. Set it to `dialog`.

To change the **OK** button into a **`Print`** button is easy. Click on that button, and the **`Properties`** window will show that you can choose which stock button it should be. Select **`Print`** from the drop-down list.

Next, we want to have a drop-down list (actually called an *option menu*) of print queue names with a label beside it prompting the user to select the queue they want. Just two widgets side by side, so we need a horizontal box. Once you have placed the horizontal box in the dialog window, adjust it using the **`Properties`** window. It will stretch the widgets we put into it to fill the vertical space, and that will look strange with an option menu, so turn that off by switching to the **`Packing`** tab and changing **Expand** to `No`.

The label should go on the left and say `Queue:`. To make the dialog a bit less cluttered-looking, set the X and Y padding of the label to 5 (see lower down on the **`Properties`** window for the label).

Finally the option menu (first widget on the 4th row of the palette) goes into the last remaining space. Give it the name `optionmenu` and change the border width to 5. The Glade work is done~~sa~~ ve it as `print.glade`. Illustration 6 shows the widget tree that you should have (from **`Show Widget Tree`**).

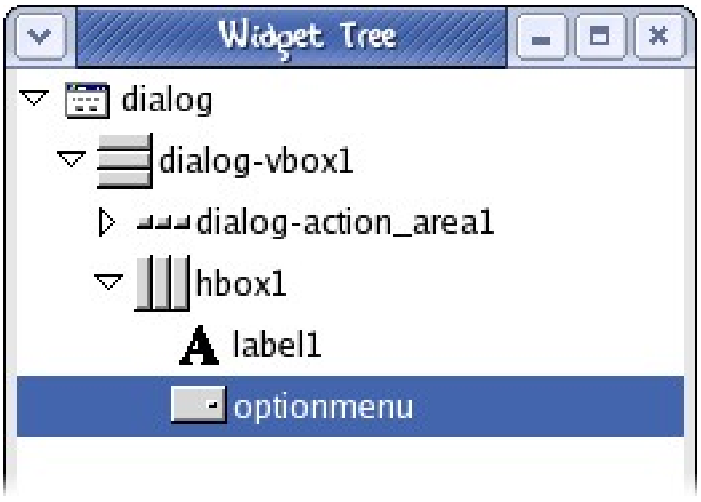


Illustration 6 Widget tree for print.glade

A short note about using Glade: when you want to select a widget but keep getting the wrong one when clicking on it, it will be because some widgets contain others—the dialog widget contains all the others in this example. You can use the right button on the mouse to get a context menu, and this will show you the widgets at the cursor position.

The Python code to drive the user interface is shown in Illustration 7. It expects the `print.glade` file to be in `/usr/local/share/print`, so make that directory using `mkdir` as root and copy `print.glade` there (or alternatively change the pathname in the Python file). Then create the

Python file as `print` (for instance, in `/usr/local/bin` and make it executable (`chmod a+x print`).

The first line, “`#!/usr/bin/python`”, makes the program executable in its own right. Instead of having to run `python /usr/local/bin/print`, you can run `/usr/local/bin/print` directly and the Python interpreter will run.

```
#!/usr/bin/python
import gtk
import gtk.glade
import os
import pycups

# Discover queues
queues = pycups.get_queues ()
queuenames = queues.keys ()
queuenames.sort ()

# Put them in a menu
menu = gtk.Menu ()
for queue in queuenames:
    menuitem = gtk.MenuItem (queue)
    menu.add (menuitem)
    menuitem.show ()
    menuitem.set_sensitive (gtk.TRUE)

# Load the Glade file
glade = "/usr/local/share/print/print.glade"
xml = gtk.glade.XML (glade)

# Attach the menu to our option menu
optionmenu = xml.get_widget ("optionmenu")
optionmenu.set_menu (menu)

# Run the dialog
dialog = xml.get_widget ("dialog")
response = dialog.run ()

if response == gtk.RESPONSE_OK:
    which = optionmenu.get_history ()
    queue = queuenames[which]
    argv = [ "lpr", "-P%s" % queue ]
    os.execvp (argv[0], argv)
```

Illustration 7 Print dialog application

A smaller program

Here is another small example program to try. It is for helping to solve crosswords by finding words that fit the letters you already have. The hard work will be done by the **grep** command; this is just to make a nice-looking interface for it. If your word looks like `---P-E` you can use grep to find words that match with:

```
grep ^...p.e$ /usr/share/dict/words
```

The task of this example program then, will be to ask the user for the word pattern, run **grep**, and display the results. Rather than using a dialog window this time, we will use a normal window. We will need to pass control to the GTK+ main loop ourselves this time. Illustration 8 shows the Glade widget tree for our crossword project, and you can see what the finished application will look like in Illustration 9.

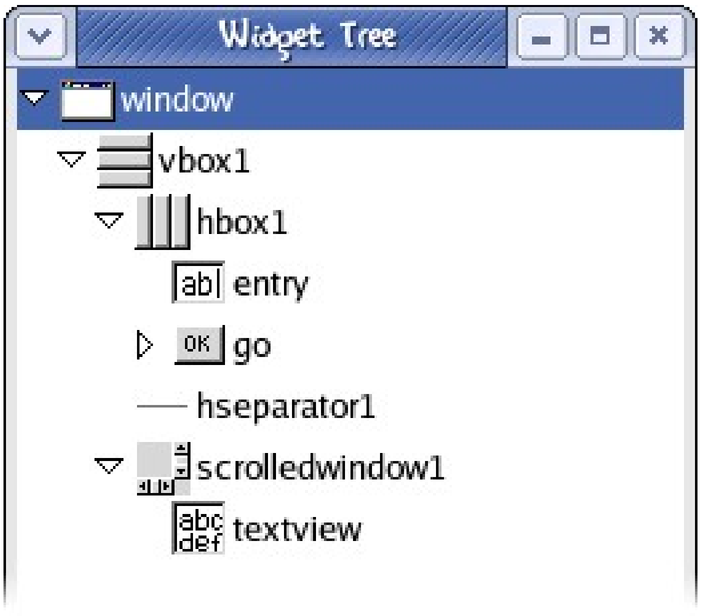


Illustration 8 Crossword widget tree

Make sure to give the widgets the correct names: the window is named “window”, the text entry box is named “entry”, the button is named “go” and the text view box is named “textview”.

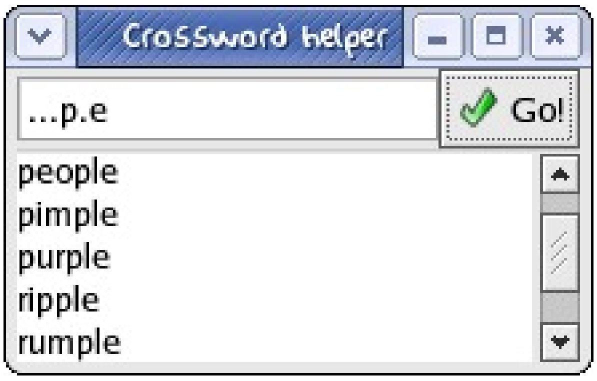


Illustration 9 Finished crossword helper

As well as putting all the widgets in place there are some properties to change in order to get them to behave as we want them to. First of all, a small visual thing: the scrolled window should have its horizontal and vertical scroll policies set to automatic. Without doing this the scroll bars would both be present all the time.

Select the button widget in Glade and look at the **`Signals`** tab on the **`Properties`** window. This is where you can associate an action with pressing the button. When you click the “...” next to the **`Signal`** field you will see the signals that the button can generate: the one we are interested in is “`clicked`”, and we want to name its handler “`on_go_clicked`”. Click **Add** to add this association.

If the user presses the Return key we want it to do the same thing as clicking on the “Go!” button. To do this we need to add the Return key as a keyboard accelerator for the button, and this can be done by clicking on **`Edit...`** (next to **`Accelerators`**) at the bottom of the **`Common`** tab. All you need to do here is associate the Return key with the “`clicked`” signal.

On to the Python code. This time we will make a *class* and define methods for it. This should be somewhat familiar from other object-orientated languages. The class will have a method called **__init__**, which is a special name and will be called when the class is brought into existence (on the very last line).

The complete Python program for our crossword project is in Illustration 10.

```
#!/usr/bin/python
import gtk.glade
import gtk
import os

class helper:
    def __init__ (self):
        xml = gtk.glade.XML ("cross.glade")
        w = xml.get_widget ('window')
        w.connect ('destroy', self.quit)
        self.entry = xml.get_widget ('entry')
        self.buffer = xml.get_widget ('textview').get_buffer ()
        xml.signal_connect ('on_go_clicked', self.on_go_clicked)
        gtk.mainloop ()

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

    def on_go_clicked (self, button):
        word = self.entry.get_text ()
        os.environ['WORD'] = word
        words = os.popen ('grep "^${WORD}$" /usr/share/dict/words').read ()
        self.buffer.set_text (words)

helper()
```

Illustration 10 Crossword program

Real world use

There are real programs that use PyGTK. Most of the Red Hat Linux configuration tools use it, for example. When you run a tool from the Red Hat menu under "Syst em Settings" there is a strong chance that it is running a Python script. The Red Hat Linux Update Agent, **up2date**, is written in Python, as is the program that displays the little RHN icon (blue tick) on the desktop panel. Take a look at the source code in `/usr/share/redhat-*/*.py`.

Even the Red Hat Linux installer uses it. When you install Red Hat Linux, Python is running from the moment you see the first graphical screen, all the way through choosing which packages you want, setting the time zone, adding users and installing the packages.

Of course it is not just Red Hat Linux that uses Python with GTK+. The GRAMPS genealogy package (at gramps.sourceforge.net) uses it too, and is written almost exclusively in Python (the one small C extension is to do with translations, and will be removed soon).

Version 2.0.0 of PyGTK was released in September 2003, and the official website for it is http://www.daa.com.au/~james/software/pygtk/.