CHAPTER 12

Graphical User Interfaces

In this chapter, you learn how to make graphical user interfaces (GUIs) for your Python programs—you know, windows with buttons and text fields and stuff like that. Pretty cool, huh?

Plenty of so-called "GUI toolkits" are available for Python, but none of them is recognized as *the* standard GUI toolkit. This has its advantages (greater freedom of choice) and drawbacks (others can't use your programs unless they have the same GUI toolkit installed). Fortunately, there is no conflict between the various GUI toolkits available for Python, so you can install as many different GUI toolkits as you want.

This chapter gives a brief introduction to one of the most mature cross-platform GUI toolkits for Python, called wxPython. For a more thorough introduction to wxPython programming, consult the official documentation (`http://wxpython.org`). For some more information about GUI programming, see Chapter 28.

A Plethora of Platforms

Before writing a GUI program in Python, you need to decide which GUI platform you want to use. Simply put, a platform is one specific set of graphical components, accessible through a given Python module, called a GUI toolkit. As noted earlier, many such toolkits are available for Python. Some of the most popular ones are listed in Table 12-1. For an even more detailed list, you could search the Vaults of Parnassus (`http://py.vaults.ca/`) for the keyword "GUI." An extensive list of toolkits can also be found in the Python Wiki (`http://wiki.python.org/moin/GuiProgramming`), and Guilherme Polo has written a paper comparing four major platforms.[1]

Table 12-1. *Some Popular GUI Toolkits Available for Python*

Package	Description	Web Site
Tkinter	Uses the Tk platform. Readily available. Semistandard.	`http://wiki.python.org/moin/TkInter`
wxPython	Based on wxWindows. Cross-platform. Increasingly popular.	`http://wxpython.org`

Continued

1. "PyGTK, PyQt, Tkinter and wxPython comparison," *The Python Papers*, Volume 3, Issue 1, pages 26–37. Available from `http://pythonpapers.org`.

Table 12-1. *Continued*

Package	Description	Web Site
PythonWin	Windows only. Uses native Windows GUI capabilities.	`http://starship.python.net/crew/mhammond`
Java Swing	Jython only. Uses native Java GUI capabilities.	`http://java.sun.com/docs/books/tutorial/uiswing`
PyGTK	Uses the GTK platform. Especially popular on Linux.	`http://pygtk.org`
PyQt	Uses the Qt platform. Cross-platform.	`http://wiki.python.org/moin/PyQt`

So which GUI toolkit should you use? It is largely a matter of taste, although each toolkit has its advantages and drawbacks. Tkinter is sort of a *de facto* standard because it has been used in most "official" Python GUI programs, and it is included as a part of the Windows binary distribution. On UNIX, however, you need to compile and install it yourself. I'll cover Tkinter, as well as Java Swing, in the section "But I'd Rather Use . . ." later in this chapter.

Another toolkit that is gaining in popularity is wxPython. This is a mature and feature-rich toolkit, which also happens to be the favorite of Python's creator, Guido van Rossum. We'll use wxPython for this chapter's example.

For information about PythonWin, PyGTK, and PyQt, check out the project home pages (see Table 12-1).

Downloading and Installing wxPython

To download wxPython, simply visit the download page, `http://wxpython.org/download.php`. This page gives you detailed instructions about which version to download, as well as the prerequisites for the various versions.

If you're running Windows, you probably want a prebuilt binary. You can choose between one version with Unicode support and one without; unless you know you need Unicode, it probably won't make much of a difference which one you choose. Make sure you choose the binary that corresponds to your version of Python. A version of wxPython compiled for Python 2.3 won't work with Python 2.4, for example.

For Mac OS X, you should again choose the wxPython version that agrees with your Python version. You might also need to take the OS version into consideration. Again, you may need to choose between a version with Unicode support and one without; just take your pick. The download links and associated explanations should make it perfectly clear which version you need.

If you're using Linux, you could check to see if your package manager has wxPython. It should be present in most mainstream distributions. There are also RPM packages for various flavors of Linux. If you're running a Linux distribution with RPM, you should at least download the wxPython common and runtime packages; you probably won't need the devel package. Again, choose the version corresponding to your Python version and Linux distribution.

If none of the binaries fit your hardware or operating system (or Python version, for that matter), you can always download the source distribution. Getting this to compile might require downloading other source packages for various prerequisites. You'll find fairly detailed explanations on the wxPython download page.

Once you have wxPython itself, I strongly suggest that you download the demo distribution, which contains documentation, sample programs, and one very thorough (and instructive) demo program. This demo program exercises most of the wxPython features, and lets you see the source code for each portion in a very user-friendly manner—definitely worth a look if you want to keep learning about wxPython on your own.

Installation should be fairly automatic and painless. To install Windows binaries, simply run the downloaded executables (`.exe` files). In OS X, the downloaded file should appear as if it were a CD-ROM that you can open, with a `.pkg` you can double-click. To install using RPM, consult your RPM documentation. Both the Windows and Mac OS X versions will start an installation wizard, which should be simple to follow. Simply accept all default settings, keep clicking Continue, and, finally, click Finish.

To see whether your installation works, you could try out the wxPython demo (which must be installed separately). In Windows, it should be available in your Start menu. When installing it in OS X, you could simply drag the wxPython Demo file to Applications, and then run it from there later. Once you've finished playing with the demo (for now, anyway), you can get started writing your own program, which is, of course, much more fun.

Building a Sample GUI Application

To demonstrate using wxPython, I will show you how to build a simple GUI application. Your task is to write a basic program that enables you to edit text files. We aren't going to write a full-fledged text editor, but instead stick to the essentials. After all, the goal is to demonstrate the basic mechanisms of GUI programming in Python.

The requirements for this minimal text editor are as follows:

- It must allow you to open text files, given their file names.
- It must allow you to edit the text files.
- It must allow you to save the text files.
- It must allow you to quit.

When writing a GUI program, it's often useful to draw a sketch of how you want it to look. Figure 12-1 shows a simple layout that satisfies the requirements for our text editor.

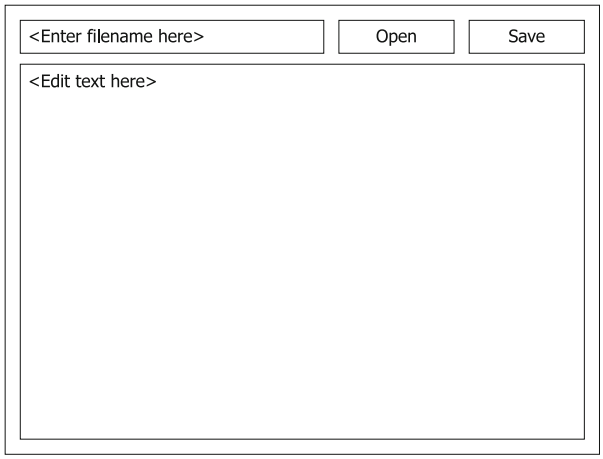

Figure 12-1. *A sketch of the text editor*

The elements of the interface can be used as follows:

- Type a file name in the text field to the left of the buttons and click Open to open a file. The text contained in the file is put in the text field at the bottom.
- You can edit the text to your heart's content in the large text field.
- If and when you want to save your changes, click the Save button, which again uses the text field containing the file name, and writes the contents of the large text field to the file.
- There is no Quit button. If you close the window, the program quits.

In some languages, writing a program like this is a daunting task, but with Python and the right GUI toolkit, it's really a piece of cake. (You may not agree with me right now, but by the end of this chapter, I hope you will.)

Getting Started

To get started, import the `wx` module:

```
import wx
```

There are several ways of writing wxPython programs, but one thing you can't escape is creating an application object. The basic application class is called `wx.App`, and it takes care of all kinds of initialization behind the scenes. The simplest wxPython program would be something like this:

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

■ **Note** If you're having trouble getting `wx.App` to work, you may want to try to replace it with `wx.PySimpleApp`.

Because there are no windows the user can interact with, the program exits immediately.

As you can see from this example, the methods in the `wx` package are written with an initial uppercase character, contrary to common practice in Python. The reason for this is that the method names mirror method names from the underlying C++ package, `wxWidgets`. Even though there is no formal rule against initial cap method or function names, the norm is to reserve such names for classes.

Windows and Components

Windows, also known as *frames*, are simply instances of the `wx.Frame` class. Widgets in the `wx` framework are created with their *parent* as the first argument to their constructor. If you're creating an individual window, there will be no parent to consider, so simply use `None`, as you see in Listing 12-1. Also, make sure you call the window's `Show` method before you call `app.MainLoop`; otherwise, it will remain hidden. (You could also call `win.Show` in an event handler, as discussed a bit later.)

Listing 12-1. *Creating and Showing a Frame*

```
import wx
app = wx.App()
win = wx.Frame(None)
win.Show()
app.MainLoop()
```

If you run this program, you should see a single window appear, similar to that in Figure 12-2.

Figure 12-2. *A GUI program with only one window*

Adding a button to this frame is about as simple as it can be—simply instantiate `wx.Button`, using `win` as the parent argument, as shown in Listing 12-2.

Listing 12-2. *Adding a Button to a Frame*

```
import wx
app = wx.App()
win = wx.Frame(None)
btn = wx.Button(win)
win.Show()
app.MainLoop()
```

This will give you a window with a single button, as shown in Figure 12-3.

Figure 12-3. *The program after adding a button*

This certainly is quite rough. The window has no title, the button has no label, and you probably don't want the button to cover the entire window in this way.

Labels, Titles, and Positions

You can set the labels of widgets when you create them, by using the `label` argument of the constructor. Similarly, you can set the titles of frames by using the `title` argument. I find it most practical to use keyword arguments with the `wx` constructors, so I don't need to remember their order. You can see an example of this in Listing 12-3.

Listing 12-3. *Adding Labels and Titles with Keyword Arguments*

```
import wx

app = wx.App()
win = wx.Frame(None, title="Simple Editor")

loadButton = wx.Button(win, label='Open')

saveButton = wx.Button(win, label='Save')

win.Show()

app.MainLoop()
```

The result of running this program should be something like what you see in Figure 12-4.

Figure 12-4. *A window with layout problems*

Something isn't quite right about this version of the program: one button seems to be missing! Actually, it's not missing—it's just hiding. By placing the buttons more carefully, you should be able to uncover the hidden button. A very basic (and not very practical) method is to simply set positions and size by using the `pos` and `size` arguments to the constructors, as in the code presented in Listing 12-4.

Listing 12-4. *Setting Button Positions*

```
import wx

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))
win.Show()

loadButton = wx.Button(win, label='Open',
                       pos=(225, 5), size=(80, 25))

saveButton = wx.Button(win, label='Save',
                       pos=(315, 5), size=(80, 25))

filename = wx.TextCtrl(win, pos=(5, 5), size=(210, 25))

contents = wx.TextCtrl(win, pos=(5, 35), size=(390, 260),
                       style=wx.TE_MULTILINE | wx.HSCROLL)

app.MainLoop()
```

As you can see, both position and size are pairs of numbers. The position is a pair of x and y coordinates, while the size consists of width and height.

This piece of code has a couple other new things: I've created a couple of *text controls* (`wx.TextCtrl` objects) and given one of them a custom *style*. The default text control is a *text field*, with a single line of editable text, and no scroll bar. In order to create a *text area*, you can simply tweak the style with the `style` parameter. The style is actually a single integer, but

you don't need to specify it directly. Instead, you use bitwise OR (the pipe) to combine various style facets that are available under special names from the `wx` module. In this case, I've combined `wx.TE_MULTILINE`, to get a multiline text area (which, by default, has a vertical scroll bar), and `wx.HSCROLL`, to get a horizontal scroll bar. The result of running this program is shown in Figure 12-5.

Figure 12-5. *Properly positioned components*

More Intelligent Layout

Although specifying the geometry of each component is easy to understand, it can be a bit tedious. Doodling a bit on graph paper may help in getting the coordinates right, but there are more serious drawbacks to this approach than having to play around with numbers. If you run the program and try to resize the window, you'll notice that the geometries of the components don't change. This is no disaster, but it does look a bit odd. When you resize a window, you assume that its contents will be resized and relocated as well.

If you consider how I did the layout, this behavior shouldn't really come as a surprise. I explicitly set the position and size of each component, but didn't say anything about how they should behave when the window was resized. There are many ways of specifying this. One of the easiest ways of doing layout in `wx` is using *sizers*, and the easiest one to use is `wx.BoxSizer`.

A sizer manages the size of contents. You simply add widgets to a sizer, together with a few layout parameters, and then give this sizer the job of managing the layout of the parent component. In our case, we'll add a background component (a `wx.Panel`), create some nested `wx.BoxSizers`, and then set the sizer of the panel with its `SetSizer` method, as shown in Listing 12-5.

Listing 12-5. *Using a Sizer*

```
import wx

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))
```

```
bkg = wx.Panel(win)

loadButton = wx.Button(bkg, label='Open')
saveButton = wx.Button(bkg, label='Save')
filename = wx.TextCtrl(bkg)
contents = wx.TextCtrl(bkg, style=wx.TE_MULTILINE | wx.HSCROLL)

hbox = wx.BoxSizer()
hbox.Add(filename, proportion=1, flag=wx.EXPAND)
hbox.Add(loadButton, proportion=0, flag=wx.LEFT, border=5)
hbox.Add(saveButton, proportion=0, flag=wx.LEFT, border=5)

vbox = wx.BoxSizer(wx.VERTICAL)
vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5)
vbox.Add(contents, proportion=1,
         flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT, border=5)

bkg.SetSizer(vbox)
win.Show()

app.MainLoop()
```

This code gives the same result as the previous program, but instead of using lots of absolute coordinates, I am now placing things in relation to one another.

The constructor of the `wx.BoxSizer` takes an argument determining whether it's horizontal or vertical (`wx.HORIZONTAL` or `wx.VERTICAL`), with horizontal being the default. The `Add` method takes several arguments. The `proportion` argument sets the proportions according to which space is allocated when the window is resized. For example, in the horizontal box sizer (the first one), the `filename` widget gets all of the extra space when resizing. If each of the three had its proportion set to 1, each would get an equal share. You can set the proportion to any number.

The `flag` argument is similar to the `style` argument of the constructor. You construct it by using bitwise OR between symbolic constants (integers that have special names). The `wx.EXPAND` flag makes sure the component will expand into the allotted space. The `wx.LEFT`, `wx.RIGHT`, `wx.TOP`, `wx.BOTTOM`, and `wx.ALL` flags determine on which sides the border argument applies, and the border arguments gives the width of the border (spacing).

And that's it. I've got the layout I wanted. One crucial thing is lacking, however. If you click the buttons, nothing happens.

■ **Tip** For more information about sizers, or anything else related to wxPython, check out the wxPython demo. It has sample code for anything you might want to know about, and then some. If that seems daunting, check out the wxPython web site, `http://wxpython.org`.

Event Handling

In GUI lingo, the actions performed by the user (such as clicking a button) are called *events*. You need to make your program notice these events somehow, and then react to them. You accomplish this by binding a function to the widget where the event in question might occur. When the event does occur (if ever), that function will then be called. You link the event handler to a given event with a widget method called `Bind`.

Let's assume that you have written a function responsible for opening a file, and you've called it `load`. Then you can use that as an event handler for `loadButton` as follows:

```
loadButton.Bind(wx.EVT_BUTTON, load)
```

This is pretty intuitive, isn't it? I've linked a function to the button—when the button is clicked, the function is called. The symbolic constant `wx.EVT_BUTTON` signifies a *button event*. The `wx` framework has such event constants for all kinds of events, from mouse motion to keyboard presses and more.

■ **Note** There is nothing magical about my choice to use `loadButton` and `load` as the button and handler names, even though the button text says "Open." It's just that if I had called the button `openButton`, `open` would have been the natural name for the handler, and that would have made the built-in file-opening function `open` unavailable. While there are ways of dealing with this, I found it easier to use a different name.

The Finished Program

Let's fill in the remaining blanks. All you need now are the two event handlers, `load` and `save`. When an event handler is called, it receives a single event object as its only parameter, which holds information about what happened. But let's ignore that here, because you're only interested in the fact that a click occurred.

Even though the event handlers are the meat of the program, they are surprisingly simple. Let's take a look at the `load` function first. It looks like this:

```
def load(event):
    file = open(filename.GetValue())
    contents.SetValue(file.read())
    file.close()
```

The file opening/reading part should be familiar from Chapter 11. As you can see, the file name is found by using `filename`'s `GetValue` method (where `filename` is the small text field, remember?). Similarly, to put the text into the text area, you simply use `contents.SetValue`.

The `save` function is just as simple. It's the exact same as `load`, except that it has a `'w'` and a `write` for the file-handling part, and `GetValue` for the text area:

```
def save(event):
    file = open(filename.GetValue(), 'w')
    file.write(contents.GetValue())
    file.close()
```

And that's it. Now I simply bind these to their respective buttons, and the program is ready to run. See Listing 12-6 for the final program.

Listing 12-6. *The Final GUI Program*

```
import wx
def load(event):
    file = open(filename.GetValue())
    contents.SetValue(file.read())
    file.close()

def save(event):
    file = open(filename.GetValue(), 'w')
    file.write(contents.GetValue())
    file.close()

app = wx.App()
win = wx.Frame(None, title="Simple Editor", size=(410, 335))

bkg = wx.Panel(win)

loadButton = wx.Button(bkg, label='Open')
loadButton.Bind(wx.EVT_BUTTON, load)

saveButton = wx.Button(bkg, label='Save')
saveButton.Bind(wx.EVT_BUTTON, save)

filename = wx.TextCtrl(bkg)
contents = wx.TextCtrl(bkg, style=wx.TE_MULTILINE | wx.HSCROLL)

hbox = wx.BoxSizer()
hbox.Add(filename, proportion=1, flag=wx.EXPAND)
hbox.Add(loadButton, proportion=0, flag=wx.LEFT, border=5)
hbox.Add(saveButton, proportion=0, flag=wx.LEFT, border=5)

vbox = wx.BoxSizer(wx.VERTICAL)
vbox.Add(hbox, proportion=0, flag=wx.EXPAND | wx.ALL, border=5)
vbox.Add(contents, proportion=1,
         flag=wx.EXPAND | wx.LEFT | wx.BOTTOM | wx.RIGHT, border=5)

bkg.SetSizer(vbox)
win.Show()

app.MainLoop()
```

You can try out the editor using the following steps:

1. Run the program. You should get a window like the one in the previous runs.
2. Type something in the large text area (for example, "Hello, world!").
3. Type a file name in the small text field (for example, `hello.txt`). Make sure that this file does not already exist or it will be overwritten.
4. Click the Save button.
5. Close the editor window (just for fun).
6. Restart the program.
7. Type the same file name in the little text field.
8. Click the Open button. The text of the file should reappear in the large text area.
9. Edit the file to your heart's content, and save it again.

Now you can keep opening, editing, and saving until you grow tired of that. Then you can start thinking of improvements. How about allowing your program to download files with the `urllib` module, for example?

You might also consider using more object-oriented design in your programs, of course. For example, you may want to manage the main application as an instance of a custom application class (a subclass of `wx.App`, perhaps?), and instead of setting up your layout at the top level of your program, you could make a separate window class (a subclass of `wx.Frame`?). See Chapter 28 for some examples.

HEY! WHAT ABOUT PYW?

In Windows, you could save your GUI programs with a `.pyw` ending. In Chapter 1, I asked you to give your file this ending and double-click it (in Windows). Nothing happened then, and I promised to explain it later. In Chapter 10, I mentioned it again, and said I would explain it in this chapter. So I will.

It's no big deal, really. It's just that when you double-click an ordinary Python script in Windows, a DOS window appears with a Python prompt in it. That's fine if you use `print` and `raw_input` as the basis of your interface, but now that you know how to make GUIs, this DOS window will only be in your way. The truth behind the `.pyw` window is that it will run Python without the DOS window, which is just perfect for GUI programs.

But I'd Rather Use . . .

As you've learned, you can choose from many GUI toolkits for Python. Here, I will give you some examples from a couple of the more popular ones: Tkinter and Jython/Swing.

To illustrate these toolkits, I've created a simple example—simpler, even, than the editor example you just completed. It's just a single window containing a single button with the label

"Hello" filling the window. When you click the button, it prints out the words "Hello, world!" In the interest of simplicity, I'm not using any fancy layout features here. Here is a simple wxPython version:

```
import wx

def hello(event):
    print "Hello, world!"

app = wx.App()

win = wx.Frame(None, title="Hello, wxPython!",
               size=(200, 100))
button = wx.Button(win, label="Hello")
button.Bind(wx.EVT_BUTTON, hello)

win.Show()
app.MainLoop()
```

The resulting window is shown in Figure 12-6.

Figure 12-6. *A simple GUI example*

Using Tkinter

Tkinter is an old-timer in the Python GUI business. It is a wrapper around the Tk GUI toolkit (associated with the programming language Tcl). It is included by default in the Windows and Mac OS distributions. The following URLs may be useful:

- `http://www.ibm.com/developerworks/linux/library/l-tkprg`
- `http://www.nmt.edu/tcc/help/lang/python/tkinter.pdf`

Here is the GUI example implemented with Tkinter:

```
from Tkinter import *
def hello(): print 'Hello, world'
win = Tk() # Tkinter's 'main window'
win.title('Hello, Tkinter! ')
win.geometry('200x100') # Size 200, 100

btn = Button(win, text='Hello ', command=hello)
btn.pack(expand=YES, fill=BOTH)

mainloop()
```

Using Jython and Swing

If you're using Jython (the Java implementation of Python), packages such as wxPython and Tkinter aren't available. The only GUI toolkits that are readily available are the Java standard library packages Abstract Window Toolkit (AWT) and Swing (Swing is the most recent and considered the standard Java GUI toolkit). The good news is that both of these are automatically available so you don't need to install them separately. For more information, visit the Jython web site and look into the Swing documentation written for Java:

- `http://www.jython.org`
- `http://java.sun.com/docs/books/tutorial/uiswing`

Here is the GUI example implemented with Jython and Swing:

```
from javax.swing import *
import sys

def hello(event): print 'Hello, world! '
btn = JButton('Hello')
btn.actionPerformed = hello

win = JFrame('Hello, Swing!')
win.contentPane.add(btn)

def closeHandler(event): sys.exit()
win.windowClosing = closeHandler

btn.size = win.size = 200, 100
win.show()
```

Note that one additional event handler has been added here (`closeHandler`) because the Close button doesn't have any useful default behavior in Java Swing. Also note that you don't need to explicitly enter the main event loop because it's running in parallel with the program (in a separate thread).

Using Something Else

The basics of most GUI toolkits are the same. Unfortunately, however, when learning how to use a new package, it takes time to find your way through all the details that enable you to do exactly what you want. So you should take your time before deciding which package you want to work with (the section "A Plethora of Platforms" earlier in this chapter should give you some idea of where to start), and then immerse yourself in its documentation and start writing code. I hope this chapter has provided the basic concepts you need to make sense of that documentation.

A Quick Summary

Once again, let's review what we've covered in this chapter:

Graphical user interfaces (GUIs): GUIs are useful in making your programs more user friendly. Not all programs need them, but whenever your program interacts with a user, a GUI is probably helpful.

GUI platforms for Python: Many GUI platforms are available to the Python programmer. Although this richness is definitely a boon, choosing between them can sometimes be difficult.

wxPython: wxPython is a mature and feature-rich cross-platform GUI toolkit for Python.

Layout: You can position components quite simply by specifying their geometry directly. However, to make them behave properly when their containing window is resized, you will need to use some sort of layout manager. One common layout mechanism in wxPython is *sizers*.

Event handling: Actions performed by the user trigger *events* in the GUI toolkit. To be of any use, your program will probably be set up to react to some of these events; otherwise, the user won't be able to interact with it. In wxPython, event handlers are added to components with the `Bind` method.

What Now?

That's it. You now know how to write programs that can interact with the outside world through files and GUIs. In the next chapter, you learn about another important component of many program systems: databases.