Chapter 22

What Can wxPython Do?

wxPython is a stable, mature graphical library. As such, it has widgets for nearly everything you plan on creating. Generally speaking, if you can't do it with wxPython, you'll probably have to create a custom GUI, such as used in video games.

I won't cover everything wxPython can do for you; looking through the demonstration code that comes with wxPython will show you all the current widgets included in the toolkit. The demonstration also shows the source code in an interactive environment; you can test different ideas within the demonstration code and see what happens to the resulting GUI.

Figure 22.0.1 shows a sample program from the wxGlade website as a demonstration of how the development of GUIs are made with wxPython.

The main group of objects you will be using are the core widgets and controls (the top left window in Figure 22.0.1). These are what you will use to build your GUI. This category includes things like buttons, check boxes, radio buttons, list boxes, menus, labels, and text boxes. As before, the wxPython demo shows how to use these items.

The bottom left window in the figure is the Properties dialog. Here is where you change the settings for whatever widget you are working with at the moment. In this example, we are looking at the properties for the highlighted checkbox shown in the "canvas" window on the right (called "wxGlade: preferences" here). This canvas window is the

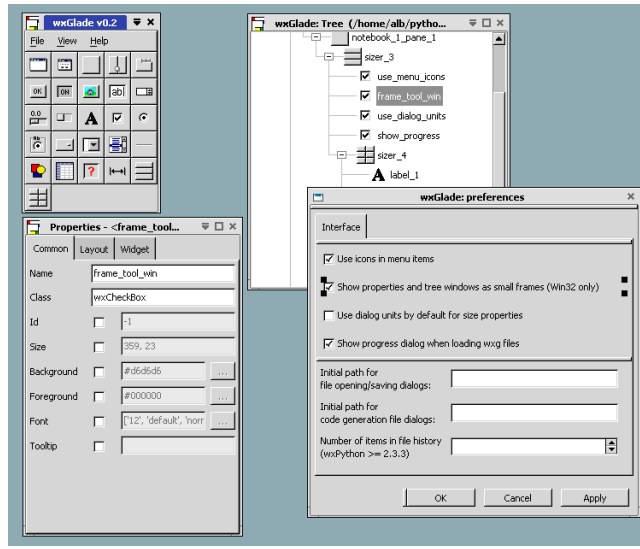

Figure 22.0.1:

visible part of the GUI you are building, i.e. this is where you actually create the user interface by adding widgets from the main wxGlade window.

Above the canvas is the "Tree" window. This is a visual hierarchy of your GUI, where you can "attach" new widgets to existing widgets. In the example, you can see that the "frame_tool_win" widget is highlighted, which is the checkbox widget in the canvas window. You'll also notice that this checkbox is nested within the "sizer_3" widget. The sizer widget is simply a container for a number of related objects, in this case the checkboxes. You'll note there is another sizer widget visible as well as a notebook widget; the notebook acts as a container for the sizer widgets.

wxPython has several standard, built-in frames and dialogs. Frames include a multiple document interface (having files of the same type contained within the parent window, rather than separate windows) and a wizard class for making simple user walk-throughs. Included dialogs range from simple "About" boxes and file selections to color pickers and print dialogs. Simple modifications to the source code

makes them plug & play-ready for your application.

There are a number of different tools shown in the wxPython demo. Most of them you will probably never use, but it's nice to know wxPython includes them and there is some source code for you to work with.

One thing I've noticed, however, is that the sample demonstrations don't always show how to best to use the widgets. For example, the wizard demo certainly displays a simple wizard with previous/next buttons. But it doesn't have any functionality, such as accepting input from the user for file names or dynamically changing the data displayed. This makes it extremely difficult to make your applications work well if you are coding off the beaten path, such as writing your program without the help of wxGlade or incorporating many different widgets into a program.

If you really want to learn wxPython, you pretty much have to either keep messing with the sample programs to figure out how they work and how to modify them to your needs, or look for a book about wxPython. Unfortunately, there are very few books on the subject and they can be hard to find. The Amazon website is probably your best bet. Alternatively, you can move to Qt, which has very extensive documentation since it is marketed towards commercial developers.

Part III

Web Programming & Game Development

Chapter 23

Developing for the Web

23.1 Choose Your Poison

When it comes to making web pages with Python, there are a variety of routes you can go. It depends on what you want to accomplish. There are full-stack frameworks, like Django, that include everything, including the kitchen sink, to build a dynamic web site from scratch. Django is written as a cohesive framework, using the same concepts throughout the entire framework so you only have to learn one way of dealing with everything.

On the other hand, TurboGears is another full-stack framework that uses best-of-breed components to accomplish its mission. Essentially, the developers looked around the Internet for the best components to include in the TurboGears project, such as an XML templating engine and a text templating engine, and then wrote the binding scripts to move data from one component to the other. This means you have to learn the idiosyncrasies of each component, even though TurboGears tries to make everything play nice.

Another popular framework is Zope. Guido van Rossum, the creator of the Python language, used to work at Zope before he moved over to Google. Zope is another all-in-one web framework like Django. Zope is also the foundation of the Plone content management system (CMS). One problem with Zope is that a number of disparate projects have developed from the original Zope code base; while they share the

same philosophies and source code, it can be difficult to figure out what you want to use and how to make it all work.

In the interest of full-disclosure, a military command I worked at was trying to develop an in-house CMS for intelligence resources and they had decided on Zope as the foundation. Unfortunately, the amount of custom code that was required to make Zope work as intended ultimately killed the project. On the plus side, the work our programmers had done with the Zope programmers ultimately produced the current version of Zope, called BlueBream. Having looked at the underlying code for Zope, it's not intuitive, in my opinion. While it is written in Python, much of the "working code" is not Python-centric, as I recall. Zope uses a lot of custom calls and special syntax to function, so even by knowing Python, you will still have to take some time getting up to speed with the "Zope way."

Alternatives to these all-in-one frameworks abound. CherryPy is a minimalist web framework that simply produces web applications, but it doesn't include HTML form filling, templating, or database functionality. You can include a variety of helper frameworks but, obviously, you will have to know how each of the components work individually so you can transfer the data between them. It's kind of like hand-making your own version of TurboGears.

23.2 Karrigell Web Framework

For purposes of this book, I will use a simple framework that combines a good amount of functionality for learning purposes. Karrigell has been around for many years but I don't think it gets the notice it should. The thing that drew me to it originally is that it allows the developer to choose how to write the underlying program. There are seven different ways you can make your source code, and you can mix-and-match as desired. All of the following items are explained more at Karrigell script styles.

1. Python scripts: these are scripts that run just like ordinary Python programs, except that *print* statements go to the client browser window instead of the console. You will need to write the HTML code for anything that will be visible by the user.

2. Karrigell Services: these are Python scripts that can handle multiple URLs, allowing a complete service with different HTML pages can be created with just one script. These are also the recommended way of developing with Karrigell.

3. Python Inside HTML: as the name says, these are HTML documents that include Python code blocks. These are functionally the same as PHP, Microsoft Active Server Pages, or Java Server Pages sites. This is good if you are coming from another web framework; you can still "write" like you are used to, only changing the functioning code from PHP, for example, to Python.

4. HTML Inside Python: just the opposite of the above, you are embedding your HTML within the rest of the Python code. This is most useful if you are writing a lot of HTML output. Rather than having to use a number of *print* statements, you can simply use regular Python quotes to output what you want; Karrigell will understand what you want and add the *print* statements at run-time.

5. HTMLTags: this is a Python module that defines a class for all valid HTML tags. When you *print* out the desired tag in the HTMLTag short-hand, Karrigell automatically outputs the correct HTML.

6. Template Engine Integration: if you desire, you can use a variety of templating engines. Template engines are much like programming languages, in that they allow for variables, functions, text replacement, etc. for the purpose of processing plain text. Essentially, they can create dynamic web pages by pulling data from a database or a template source file. XSLT is an example of a template model for processing XML files.

7. Karrigell Templates: this is a built-in template engine intended for use with Karrigell services and Python scripts. It is different from other template engines by not including many of the typical programatic controls, as these are handled by Python directly. It also supports inclusion of other templates, if desired.

23.2.1 Python Scripts

Since you have to manually write all the HTML output, you obviously need to know something about HTML. For example, to make a simple table, you would have to print out each HTML line in your Python script, as shown below:

Listing 23.1: Karrigell Python->HTML Table

```
print "<TABLE>"
print "<TR>"
print "<TD>Name</TD>"
print "<TD>Address</TD>"
print "</TR>"
print "</TABLE>"

#Or you can use Python's multi-line syntax

print """<TABLE>
        <TR>
                <TD>Name</TD>
                <TD>Address</TD>
        </TR>
</TABLE>"""
```

23.2.2 Karrigell Services

These are Python scripts that Karrigell maps to separate URLs; basically, each script becomes a new website. This is the ideal way of writing web apps for Karrigell since access to user values is straightforward and application logic is in one location.

Essentially, each function is defined at the module level to be mapped to a URL; arguments can be passed from one script to another as "?" arguments in the URL. For example, the script named *foo.ks* contains the function *bar()* and is located at the URL *foo.ks/bar*. Listing 23.2 provides some clarification:

Listing 23.2: Karrigell Services as URLs

```
#A web link is given
```

```
<a href ="script.ks/foo?bar=300">

#or a web form is submitted
<form action="script.ks/foo">
<input name = "bar">
<input type = "submit" value = "ok">

#The associated script is simple
def foo(bar):
        print bar

#To jump from one function to another, specify the function name
#in the link or form action
def index():
        print "<a href = 'foo?name=bar'>go to foo</a>"
def foo(name):
        print "<img src = '../pic.jpg'>"
        print name
```

23.2.3 Python Inside HTML

As mentioned previously, embedding Python within HTML is much like developing with PHP or other languages. In this case, the Python code is separated from the HTML with special tags: <% and %>. Everything between those special tags is normal Python code, so you can import modules, create classes and instances, work with the file system, etc. Listing 23.3 provides an example:

Listing 23.3: Embedding Python in HTML

```
The current date is
<% import datetime
print datetime.date.today().strftime("%d:%m:%y")
%>
```

23.2.4 HTML Inside Python

Embedding HTML in Python code is easier than the opposite way, since you don't have to deal with *print* statements all the time. Listing

23.4 shows the two main ways of embedding HTML:

Listing 23.4: Embedding HTML in Python

```
#For simple HTML, simply use quotes and no print statement
import os
currentDir = os.getcwd()
"Current directory is <b>"+currentDir+"</b>"

#For longer HTML blocks, use Python triple quotes
the_Pythons={'lead':'Graham Chapman',
          'oldest':'John Cleese',
      'American':'Terry Gilliam',
          'music':'Eric Idle',
          'brains':'Terry Jones',
          'writer':'Michael Palin'}
"""
<table border=1>
<tr backgroundcolor=green>
<td>One of the best comedy groups ever</td>
</tr>
</table>
<table>
"""
for item in the_Pythons.keys():
    "<tr><td>%s</td><td>%s</td></tr>"
%(item,the_Pythons[item])
"</table>"
```

23.2.5 HTMLTags

HTMLTags is a Python module that defines all valid HTML tags, allowing you to generate HTML with Python, rather than simply embedding it. All HTML tags must be written in uppercase to be recognized.

Here is the obligatory example:

Listing 23.5: Generating HTML via HTMLTags

```
head = HEAD()
```

```
head <= LINK(rel="Stylesheet",href="../doc.css")
head <= TITLE('Record collection')+stylesheet

body = BODY()
body <= H1('My record collection')

table = TABLE(Class="content")
table <= TR(TH('Title')+TH('Artist'))
for rec in records:
        table<=TR(TD(rec.title,Class="title")+TD(rec.artist,Clas
body <= table

print HTML(head+body)
```

23.2.6 Template engine integration

There isn't much I can say about template engines, other than Karrigell Templates (KT) is a built-in engine that uses Python for most of the processing operations. However, you can use KT to include other templates and specify translation strings, which are then passed to the Karrigell translation engine. To use other engines, you simply use the syntax necessary to add it to the associated Python script, just like a normal program.

23.2.7 Karrigell Templates

Karrigell Templates are designed for use with Karrigell Services and Python scripts. An example of KT is in Listing 23.6.

Listing 23.6: Karrigell Template

```
<html>
<head>
<link rel="stylesheet" href="$this.baseurl/css/my.css">
<title>MyApp $data.title</title>
</head>
<body>
@[$data.bodytmpl]
<hr>
```

```
<i>_[Powered by Karrigell]</i>
<p />
</body>
</html>
```

The body contains the special tag `@[$data.bodytmpl]`, which is actually a call to another template, which simply contains the HTML line `<h1>Welcome to $data.who home page!<h1>`.

If setup in Karrigell correctly (I haven't shown the entire process), when the template is called, it identifies what language the user's browser is set to and provides a different output based on the language, as shown in Listing 23.7.

Listing 23.7: HTML Output of Karrigell Template

```
<!--Browser set to English-->
<html>
<head>
<link rel="stylesheet" href="/css/my.css">
<title>MyApp - home</title>
</head>
<body>
<h1>Welcome to my home page!</h1>
<hr>
<i>Powered by Karrigell</i>
<p />
</body>
</html>

<!--Browser set to French-->
<html>
<head>
<link rel="stylesheet" href="/css/my.css">
<title>MyApp - home</title>
</head>
<body>
<h1>Welcome to my home page!</h1>
<hr>
<i>MotorisÃ© par Karrigell</i>
```