

SAMPLE CHAPTER



wxPython

IN ACTION

Noel Rappin
Robin Dunn

 MANNING



wxPython in Action

by Noel Rappin

and

Robin Dunn

Sample Chapter 2

Copyright 2006 Manning Publications

brief contents

PART 1 INTRODUCTION TO WXPYTHON 1

- 1 ■ Welcome to wxPython 3
- 2 ■ Giving your wxPython program a solid foundation 29
- 3 ■ Working in an event-driven environment 56
- 4 ■ Making wxPython easier to handle with PyCrust 83
- 5 ■ Creating your blueprint 116
- 6 ■ Working with the basic building blocks 146

PART 2 ESSENTIAL WXPYTHON 183

- 7 ■ Working with the basic controls 185
- 8 ■ Putting widgets in frames 224
- 9 ■ Giving users choices with dialogs 258
- 10 ■ Creating and using wxPython menus 293
- 11 ■ Placing widgets with sizers 323
- 12 ■ Manipulating basic graphical images 356

PART 3 ADVANCED WXPYTHON 391

- 13 ■ Building list controls and managing items 393
- 14 ■ Coordinating the grid control 425
- 15 ■ Climbing the tree control 460
- 16 ■ Incorporating HTML into your application 485
- 17 ■ The wxPython printing framework 504
- 18 ■ Using other wxPython functionality 521

Giving your wxPython program a solid foundation

This chapter covers

- Creating application objects
- Directing output from a wxPython program
- Shutting down a wxPython application
- Creating and using top-level window objects
- Adding objects and subwindows to a frame

The foundation of a house is a concrete structure that provides a strong base for the rest of the construction. Your wxPython program also has a foundation, made up of two required objects that support the rest of your application. These are the *application object* and the *top-level window object*. Using these objects properly will give your wxPython application a strong base to start with, and will make the job of constructing the rest of your application easier.

In this chapter, you will work with the application object to customize global aspects of your program, including initialization, redirecting output, and shut-down. You will use window objects in various styles, and put together basic widget combinations. You will also use simple default dialog boxes to get user information. At the end of the chapter, we'll help you diagnose and troubleshoot common problems with usage of application and top-level window objects.

2.1 What do I need to know about the required objects?

Let's start with a description of the two foundation objects. The application object manages the main event loop, which is the heartbeat of your wxPython program. The event loop will be covered in detail in chapter 3. For now, it is enough to say that it is the application object's job to start the main event loop. In addition, the application object has the final chance to respond to any events which are otherwise ignored by your application. Without the application object, your wxPython application cannot run.

The top-level window object generally manages the most important data and controls and presents them to the user. For example, in a word-processing program, the main window is the display of the document, and will likely manage at least some of the data for the document (depending, of course, on the exact architecture of your application). Similarly, the main window of your web browser both displays the page you are looking at and manages that page as a data object.

Figure 2.1 gives a basic schematic of the relationship between the two foundation objects and the rest of your application.

As this diagram shows, the application object “owns” both the top-level window and the main event loop. The top-level window manages the components in that window, and any other data objects you assign to it. That window and its components trigger events based on user actions, and receive event notifications to make changes in the display. In the next sections, we'll discuss the application and top-level window objects in more detail.

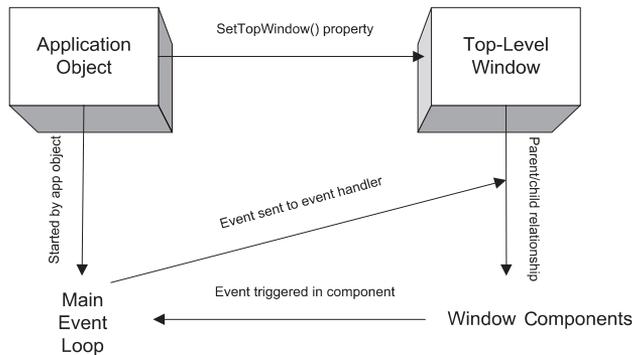


Figure 2.1 A schematic of the basic wxPython application structure, showing the relationship between the application object, the top-level window, and the main event loop

2.2 How do I create and use an application object?

Every wxPython application needs exactly one application object. The application object must be an instance of the class `wx.App`, or a custom subclass thereof. The primary purpose of the application object is to manage the main event loop behind the scenes. This event loop responds to windowing system events and dispatches them to the appropriate event handler. The application object is so important to the management of a wxPython process that you cannot create any wxPython graphical objects until after your program has instantiated an application object.

The parent `wx.App` class also defines a few properties which are global to the entire application. Much of the time, that's all the functionality you'll need from your application object. A custom application subclass can be used if you need to manage other global data or connections (such as a database connection). In some cases, you might also want to extend the main event loop for more specialized error or event handling. However, the default event loop will be suitable for nearly all wxPython applications that you will write.

2.2.1 Creating a `wx.App` subclass

Creating your own subclass of `wx.App` is so simple that it's often a good idea to create one when you start your application even if you don't need any custom functionality. That way, you'll have the subclass if you need it later. To create and use a `wx.App` subclass, you need to perform four steps:

- 1 Define the subclass.
- 2 Write an `OnInit()` method within the subclass definition.
- 3 Create an instance of the class in the main section of your program.
- 4 Call the `MainLoop()` method of the application instance. This method transfers program control to wxPython.

We saw the `OnInit()` method in chapter 1. It's called by the wxPython system when the application is started and before the main event loop begins. This method takes no parameters and returns a boolean value—if the return value is `False`, then the application will exit immediately. In most cases, you'll want to hardwire `True` as the result of this method. Exiting might be the proper way to handle certain error conditions, such as the absence of a required resource.

Because the `OnInit()` method exists, and is part of the wxPython framework, any initialization needed for your custom class is typically managed there, and not in the Python `__init__` special method. If you decide that you need an `__init__` method for some reason, you must call the `__init__` method of the parent class in that method, as in the following.

```
wx.App.__init__(self)
```

Typically, you'll create at least one frame object within the `OnInit()` method, and you'll also call the `Show()` method of that frame. You may optionally specify that the frame is the top window for the application by calling the method `SetTopWindow()`. The top window is used as the default parent for dialogs that are created without a parent—it's essentially the main window of your program. We'll discuss the top-level window in section 2.5.

When to omit a wx.App subclass

You aren't required to create your own `wx.App` subclass. You usually will want to do so to be able to create your top-level frame in the `OnInit()` method. But there is nothing stopping you from creating the frame outside of the application definition in some other part of calling script—the most common alternate place is the `__main__` clause. The only restriction is that your `wx.App` instance has to have been created first. Generally, it is only a good idea to avoid creating a `wx.App` subclass if there's just one frame in the system, and therefore the application setup is trivial. In such a case, wxPython provides the convenience class `wx.PySimpleApp`. The class provides a skeleton `OnInit()` method, and is defined as follows:

```
class PySimpleApp(wx.App):

    def __init__(self, redirect=False, filename=None,
                 useBestVisual=False, clearSigInt=True):
        wx.App.__init__(self, redirect, filename, useBestVisual,
                        clearSigInt)

    def OnInit(self):
        return True
```

It doesn't get much simpler than that. A sample usage of `wx.PySimpleApp` might look like this:

```
if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = MyNewFrame(None)
    frame.Show(True)
    app.MainLoop()
```

In the first line of this snippet, you create the application object as an instance of `wx.PySimpleApp()`. Since we're using the `wx.PySimpleApp` class, we don't have a custom `OnInit` method, so we define a frame in the second line of the snippet—since it has no parent specified, it's a top-level frame. (Obviously, the `MyNewFrame` class needs to be defined somewhere.) The third line of the code shows the frame, and the last line calls the application main loop, and we're good to go.

As you can see, using `wx.PySimpleApp` allows you to run your wxPython program without creating your own custom application class. You should only use `wx.PySimpleApp` if the application is, well, simple, and doesn't need any other global parameters.

NOTE *Naming Conventions*—While wxPython does a fantastic job of simplifying a complex C++ toolkit, the C++ origins of the tool do leak through in spots. One of the most noticeable examples of the C++ heritage has to do with naming conventions. In Python, method names usually use the `lower_case_separated_by_underscores` or the `lowerCaseInterCap` style. However, the C++ convention which wxWidgets uses for methods is the `UpperCaseInterCap` style. This can be jarring if you are used to the Python style. For consistency's sake, it is recommended that you use the wxWidgets style in your wxPython classes. (Of course, you'll need to use it if you want to override wxWidgets methods).

Also note that the wxPython classes use explicit `Get` and `Set` methods for properties. That's more of a C++ style because most Python programs wouldn't define special accessor methods for simple cases.

The data members of the C++ classes are private—in most cases you must access the data of a wxPython class by using the access methods, you cannot use bare attribute names.

2.2.2 Understanding the application object lifecycle

The lifecycle of your wxPython application object begins when the application instance is created and ends just after the last application window is closed. This does not necessarily correspond to the beginning and ending of the Python script that surrounds your wxPython application. The script may choose to do some activity before creating the wxPython application, and may do further cleanup after the application `MainLoop()` exits. All wxPython activity, however, must be performed during the life of the application object. As we've mentioned, this means that your main frame object cannot be created until after the `wx.App` object is created. (This is one reason why we recommend creating the top-level frame in the `OnInit()` method—doing so guarantees that the application already exists.)

As figure 2.2 shows, creating the application object triggers the `OnInit()` method and allows new window objects to be created. After `OnInit()`, the script calls `MainLoop()`, signifying that wxPython events are now being handled. The application continues on its merry way, handling events until the windows are closed. After all top-level windows are closed, the `MainLoop()` function returns to the calling scope and the application object is destroyed. After that, the script can close any other connections or threads that might exist.

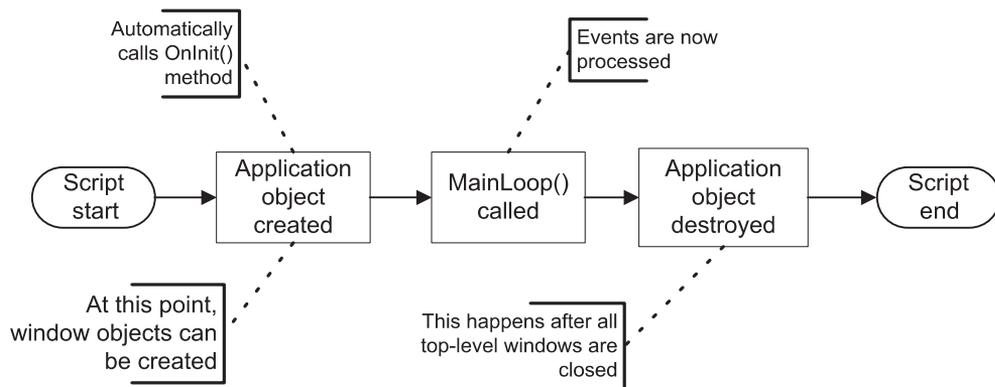


Figure 2.2 Major events in the wxPython application lifecycle, including the beginning and ending of both the wxPython application and the script which surrounds it

One reason to be aware of the main application life cycle is that, while active, a wxPython application will take control of certain system functions, such as the standard output streams. We'll discuss how to direct output in the next section

2.3 How do I direct output from a wxPython program?

All Python programs can output text via two standard streams: the standard output stream, `sys.stdout`, and the standard error stream `sys.stderr`. An ordinary Python script directs the standard output streams back to the console from which the script was launched. However, when your application object is created you can decide to have wxPython take control of the standard streams and redirect the output to a window instead. This redirect behavior is the default behavior for wxPython under Windows. In Unix systems, however, where there is more likely to be an actual console window, wxPython does not control the standard streams by default. In all systems the redirection behavior can be explicitly specified when the application object is created. We recommend taking advantage of this feature and always specifying redirect behavior to avoid any problems from different behavior on different platforms.

2.3.1 Redirecting output

If wxPython is controlling the standard streams, then text sent to the streams via any mechanism—including a print statement or a system traceback—is redirected to a separate wxPython frame. Text sent to the streams before the wxPython application begins or after it ends is, of course, processed normally. Listing 2.1, demonstrates both the application lifecycle and the `stdout/stderr` redirection.

Listing 2.1 A sample startup script showing output stream redirection

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

import wx
import sys

class Frame(wx.Frame):

    def __init__(self, parent, id, title):
        print "Frame __init__"
        wx.Frame.__init__(self, parent, id, title)

class App(wx.App):
```

```

def __init__(self, redirect=True, filename=None):
    print "App __init__"
    wx.App.__init__(self, redirect, filename)

def OnInit(self):
    print "OnInit"    ← Writing to stdout
    self.frame = Frame(parent=None, id=-1, title='Startup') ← Creating the frame
    self.frame.Show()
    self.SetTopWindow(self.frame)
    print ">> sys.stderr, \"A pretend error message\"    ← Writing to stderr
    return True

def OnExit(self):
    print "OnExit"

if __name__ == '__main__':
    app = App(redirect=True)    ❶ Text redirection starts here
    print "before MainLoop"
    app.MainLoop()    ❷ The main event loop is entered here
    print "after MainLoop"

```

❶ This line creates the application object. After this line, all text sent to `stderr` or `stdout` can be redirected to a frame by wxPython. The arguments to the constructor determine whether this redirection takes place.

❷ When run, this application creates a blank frame, and also generates a frame with the redirected output, as shown in figure 2.3.

Notice also that both `stdout` and `stderr` messages get directed to the window.

After you run this program you'll see that your console has the following output:

```

App __init__
after MainLoop

```

The first line is generated before the frames are opened, the second line is generated after they close.

By looking at both the console and the output frame, we can trace the application lifecycle.

The first bubble in figure 2.2—Start Script—corresponds to the first lines run from the script's `__main__` clause. The transition to the next bubble comes immediately in the line marked ❶. The instantiation of the instance calls the method

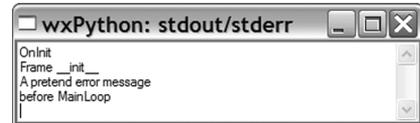


Figure 2.3 The `stdout/stderr` window created by Listing 2.1

`wx.App.__init__()`. Then control goes to `OnInit()`, which is automatically called by wxPython. From there, the program jumps to the `wx.Frame.__init__()`, which is run when the `wx.Frame` instance is instantiated. Finally, control winds back to the `__main__` clause, where `MainLoop()` is invoked, corresponding to the third bubble in figure 2.2. After the main loop ends, then `wx.App.OnExit()` is called by wxPython, transitioning to the fourth bubble, and then the rest of the script finishes out the process.

“Wait a minute,” you say, “the message from `OnExit()` didn’t display in either the window or the console.” As we’ll see, the message does display in the wxPython frame, but it does so right before the window is closed, so that it’s nearly impossible to capture in a screen shot.

The quickly vanishing `OnExit()` message is a symptom of a larger issue with the output frame. Although it’s a useful feature during development, you don’t necessarily want the error stream frame popping out in a user’s face at run time. Furthermore, if an error condition happens during the `OnInit()` method, it gets sent to the output frame, but the error causes the application to exit, since `OnInit()` will return a `False` value in case of an error condition. The result is that the line of text is displayed but disappears far too quickly to be seen.

2.3.2 Modifying the default redirect behavior

In order to modify this behavior, wxPython allows you to set two parameters when creating the application. The first parameter, `redirect`, causes output to be redirected to the frame if it is `True`. If the value is `False`, output stays at the console. If the `redirect` parameter is `True`, then the second parameter, `filename`, can also be set. If so, output is redirected to a file with that name, rather than to the wxPython frame. Therefore, changing the `wx.App` creation at annotation ❶ in listing 2.1 to:

```
app = App(False)
```

causes all of the following output to go to the console:

```
App __init__
OnInit
Frame __init__
A pretend error message
before MainLoop
OnExit
after MainLoop
```

Notice that the `OnExit()` message is displayed here. Changing the line to:

```
app = App(True, "output")
```

will cause all the redirected lines to be sent to a file named `output`. The `App` `__init__` and after `MainLoop` messages will still be sent to the console, however, because they occur outside of the time period where the `wx.App` object has control of the streams.

2.4 How do I shut down my wxPython application?

When the last top-level window in your application is closed by the user, the wxPython application exits. By top-level, we mean any frame without a parent, and not just the frame designated using the `SetTopWindow()` method. This includes any frames created by wxPython itself. For instance, in listing 2.1, the application does not exit until *both* the main frame and the output redirection frame are closed, even though only the main frame is registered using `SetTopWindow()`, and even though the application doesn't explicitly create the output redirect frame. To trigger a shutdown programatically, you can call `Close()` on all top-level windows.

2.4.1 Managing a normal shutdown

During the shutdown process, wxPython takes care of deleting all its windows and freeing their resources. You have one hook into the exit process to perform your own cleanup. If defined, the `OnExit()` method of your `wx.App` subclass is called after the last window closes but before wxPython's internal cleanup. You can use this method to clean up any non-wxPython resources you've created (a database connection, for example). Even if the application is closed with `wx.Exit()`, the `OnExit()` method is still triggered.

If for some reason you want the application to continue after the last window closes, you can change the default behavior using the `wx.App` method `SetExitOnFrameDelete(flag)`. If the `flag` parameter is set to `False`, the program will continue to run even after the last window closes. This means that the `wx.App` instance will continue to live, and the event loop will continue to process events. You could, for example, then create all new top-level windows. The application will remain alive until the global function `wx.Exit()` is explicitly called.

A further subtlety is that wxPython will not trigger the shutdown process before the main loop is even entered. Specifically, if you open a dialog in your `OnInit()` method, you can close it without fear that wxPython will interpret that as closing your last top-level window and shut itself down.

2.4.2 Managing an emergency shutdown

You can't always close your program in a controlled way. Sometimes, you need to end the application immediately and you don't care that your program cleans up after itself fully. For example, a critical resource may have closed or become corrupted. If the system is shutting down, you may not be able to do all the cleanup.

There are two mechanisms for exiting your wxPython application in an emergency situation. You can call the `wx.App` method `ExitMainLoop()`. This method explicitly causes the main message loop to break, causing the control to leave the `MainLoop()` function. This will generally end the application—it's effectively equivalent to closing all the top-level windows.

You can also call the global method `wx.Exit()`. Neither method is recommended for normal use because it may cause some cleanup functions to be skipped.

Sometimes, your application will need to shut down due to an event outside of its control. An example of this is when the underlying operating system is about to shut down or log the user off. In that case, your application gets one shot at cleanup to save documents or close connections or whatever. If your application binds an event handler for the `wx.EVT_QUERY_END_SESSION` event, then that event handler is called when wxPython is notified of the shutdown. (We'll show how to bind events to handlers later in the chapter, and in more detail in chapter 3.) The event parameter is a `wx.CloseEvent`. The close event may allow the application to veto the shutdown. Use the event method `CanVeto()` to find out. The application can make its veto known by calling the event method `Veto()`. You might want to do this if you cannot successfully save or close all resources. The default handler for the `wx.EVT_QUERY_END_SESSION` event calls the `Close()` method of the top-level windows, which will in turn send the `wx.EVT_CLOSE` event to the top-level windows giving you another chance to control the shutdown process. If any of the `Close()` methods returns `False` then the application attempts to veto the shutdown.

2.5 How do I create and use the top-level window object?

A top-level window object is a widget (usually a frame) that is not contained by another widget in your application—it's what a typical user would point to and say, "That's the program." The top-level window object is usually the main window of your application and contains widgets and interface objects that the user interacts with. As we have seen, the application exits when all top-level windows are closed.

Your application must have at least one top-level window object. The top-level window object is usually a subclass of the class `wx.Frame`, although it can also be a subclass of `wx.Dialog`. Most of the time, you will define custom subclasses of `wx.Frame` for use in your application. However, there are a number of pre-defined `wx.Dialog` subclasses that provide many of the typical dialogs that you might encounter in an application.

There's some naming confusion here, due to overloading of the word *top*. A generic “top-level” window is *any* widget in your application that doesn't have a parent container. Your application must have at least one of these, but it can have as many as you'd like. Only one of these windows, however, can be explicitly blessed by wxPython as the main top window by using `SetTopWindow()`. If you do not specify a main window with `SetTopWindow`, then the first frame in the `wx.App`'s top-level window list is considered to be the top window. So, explicitly specifying the top window is not always necessary—you don't need to if, for example, you only have one top window. Repeated calls to `SetTopWindow()` will replace the current top window—an application can only have one top window at a time.

2.5.1 Working with `wx.Frame`

In wxPython parlance, a frame is the name given to what a GUI user normally calls a window. That is to say, a frame is a container that the user can generally move freely around on the screen, and which often includes such decorations as a title bar, menubar, and resize targets in the corners. The class `wx.Frame` is the parent class of all frames in wxPython. There are also a few specialized subclasses of `wx.Frame` that you may use. This section will give an overview of the `wx.Frame` family—enough for you to get started using them. A more complete description of the `wx.Frame` class will be presented in chapter 8.

When you create subclasses of `wx.Frame`, the `__init__()` method of your class should call the parent constructor `wx.Frame.__init__()`. The signature of that constructor is as follows.

```
wx.Frame(parent, id=-1, title="", pos=wx.DefaultPosition,
         size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE,
         name="frame")
```

This constructor takes several parameters. In normal use, however, at least some of the defaults are reasonable options. We will see parameters similar to this constructor again and again in other widget constructors—it's a very similar pattern in wxPython. Table 2.1 describes each of the parameters.

Remember, these are the parameters as passed to the parent constructor method, `wx.Frame.__init__()`. The argument signature of the constructor to your

Table 2.1 Parameters of the wx.Frame constructor method

Parameter	Description
parent	The parent window of the frame being created. For top-level windows, the value is None. If another window is used for the parent parameter then the new frame will be owned by that window and will be destroyed when the parent is. Depending on the platform, the new frame may be constrained to only appear on top of the parent window. In the case of a child MDI window, the new window is restricted and can only be moved and resized within the parent.
id	The wxPython ID number for the new window. You can pass one in explicitly, or pass -1 which causes wxPython to automatically generate a new ID. See the section “Working with wxPython ID” for more information.
title	The window title—for most styles, it’s displayed in the window title bar.
pos	A wx.Point object specifying where on the screen the upper left-hand corner of the new window should be. As is typical in graphics applications, the (0, 0) point is the upper left corner of the monitor. The default is (-1, -1), which causes the underlying system to decide where the window goes. See the section “Working with wx.Size and wx.Point” for more information.
size	A wx.Size object specifying the starting size of the window. The default is (-1, -1), which causes the underlying system to determine the starting size. See the section “Working with wx.Size and wx.Point” for more information.
style	A bitmask of constants determining the style of the window. You may use the bitwise or operator () to combine them when you want more than one to be in effect. See the section “Working with wx.Frame styles” for usage guidelines.
name	An internal name given to the frame, used on Motif to set resource values. Can also be used to find the window by name later.

class can, and often will, be different. This allows you to conveniently ensure default values for your own frame by not allowing them to be modified by a call to your constructor. For example, you might want your frame class to always be a 300 pixel square. In that case, you probably wouldn’t have a size argument in your class constructor, but would just explicitly pass (300, 300) to the wx.Frame.__init__() method. Listing 2.2 shows a frame class that does not allow any of the attributes of the window to be passed as an argument.

Listing 2.2 A frame subclass which sets its own defaults

```
class MyFrame(wx.Frame):
    def __init__(self):
        wx.Frame.__init__(self, None, -1, "My Friendly Window",
                           (100, 100), (100, 100))
```

In listing 2.2, the `__init__()` method of `MyFrame` does not take any arguments. This means that `MyFrame` users cannot override the hardwired arguments that `MyFrame.__init__()` passes to the superclass `wx.Frame.__init__()`. Remember that a determined user of your class can always change the default values by calling setter methods after the frame is instantiated.

2.5.2 Working with wxPython IDs

Table 2.1 lists the *wxPython ID* number of the new frame. ID numbers are a feature of all widgets in wxPython, and it's worth a few paragraphs to explain how they work. Every widget in a wxPython application has a window identifier. The ID numbers must be unique within each frame, but you can reuse ids between frames. We recommend, however, that you try to have your ID numbers be unique across your application, to prevent errors and confusion when processing events. However, there are few standard predefined ID numbers in wxPython, which have specific meanings within parts of the code (for example, `wx.ID_OK` and `wx.ID_CANCEL` are assumed to be the ID numbers of the OK and Cancel buttons in a dialog box). It's usually not a problem to reuse standard ID numbers in your application as long as you use them in the manner expected. The ID number is usually the second parameter in the constructor of a wxPython widget object, after the parent widget. The most important use of ID numbers in wxPython is to create a unique relationship between an event that happens to a specific object and a function which is called in response to that event. Using a duplicate ID can cause the wrong function to be triggered in response to an event.

There are three ways to create the ID numbers used by a widget:

- 1 Explicitly pass a positive integer into the constructor
- 2 Get wxPython to create IDs for you using the `wx.NewId()` function
- 3 Pass either the global constant `wx.ID_ANY` or `-1` to a widget constructor

Explicitly choosing the ID

First and most straightforwardly, you can explicitly pass a positive integer into the constructor—that number becomes the widget's ID. If you pass a specific number, it is your responsibility to make sure that you do not duplicate ID numbers within a frame or reuse one of the predefined constants. You can ensure that wxPython does not use your explicit ID elsewhere in the application by calling the global function `wx.RegisterId()`. To prevent your program from duplicating wxPython

IDs, you should avoid using ID numbers between the global constants `wx.ID_LOWEST` and `wx.ID_HIGHEST`.

Using the global `NewID()` function

However, ensuring the uniqueness of ID numbers can become burdensome quickly. You can instead get wxPython to create ids for you using the `wx.NewId()` function:

```
id = wx.NewId()
frame = wx.Frame.__init__(None, id)
```

Using a constant to say ‘I don’t care’

Alternately, you can pass either the global constant `wx.ID_ANY` or `-1` to many widget constructors, and then wxPython will generate the new ID for you. Then you can use the `GetId()` method if you need the ID later on:

```
frame = wx.Frame.__init__(None, -1)
id = frame.GetId()
```

There’s no particular functional difference between the styles.

2.5.3 Working with `wx.Size` and `wx.Point`

The list of `wx.Frame` constructor arguments (table 2.1) also referenced the classes `wx.Size` and `wx.Point`. These two classes will be used frequently in your wxPython programming. Because of that, some Python-specific shortcuts have been added to the classes.

The `wx.Size` and `wx.Point` classes are quite similar both in their wxWidgets method list and in how they are represented in wxPython. The `wx.Point` class, surprisingly enough, represents a point or position. The constructor takes two arguments for the *x* and *y* values of the point. The values both default to zero if not set:

```
point = wx.Point(10, 12)
```

To set both dimensions in one line, use the function `Set(x,y)`. To retrieve them both at once, use `Get()`, which returns the two values as a Python tuple. Unlike most wxWidgets classes, `wx.Point` has functionality defined to retrieve the *x* and *y* values as simple Python properties such that the *x* and *y* values are accessible like ordinary Python attributes:

```
x = point.x
y = point.y
```

In addition, `wx.Point` instances can be transparently added, subtracted, and compared just like other Python objects. For example:

```
x = wx.Point(2, 3)
y = wx.Point(5, 7)
z = x + y
bigger = x > y
```

In a `wx.Point` instance, the attributes are expected to be integers. If you need floating-point coordinates, you can use the class `wx.RealPoint`, which works much the same way as `wx.Point`.

The `wx.Size` class is almost identical to `wx.Point`, except that the instance variables are named `width` and `height` instead of `x` and `y`. Otherwise, it has the same attribute and operator features defined.

When a `wx.Point` or `wx.Size` instance is needed anywhere in your wxPython program—for example, in the constructor for another object—you do not need to create the instance explicitly. Instead, you can pass a Python tuple to the constructor, and wxPython will implicitly create the `wx.Point` or `wx.Size` instance:

```
frame = wx.Frame(None, -1, pos=(10, 10), size=(100, 100))
```

This works not just in constructors for other objects, but also anyplace where a `wx.Point` or `wx.Size` is expected by a wrapped C++ method or function call. You can even write something like this, if you so desire:

```
frame.SetPosition((2, 3))
```

2.5.4 Working with `wx.Frame` styles

The `wx.Frame` constructor takes a bitmask as a `style` parameter. Every wxPython widget object takes a similar `style` parameter, although the exact values that are defined are different for each type of widget. This section will discuss the styles used for `wx.Frame`. At least some of this is applicable to other wxPython widgets. The widget definitions in part 2 will discuss styles applicable to each class.

WHAT'S A BITMASK?

A bitmask is a way of compactly storing information about system attributes that is especially useful when there are a limited number of attributes with boolean values and the values are more or less mutually independent. In wxPython, bitmasks are used to manage a number of different attributes throughout the framework, most notably style information.

In a bitmask, the individual attributes are assigned constant values corresponding to powers of two, and the value of the bitmask is the sum of all the attributes which are “turned on”. In binary notation, the power of two system guarantees that each attribute corresponds to a single bit in the total sum, allowing all of the attribute state to be compactly stored in

a single integer or long value. For example, if attribute `a=1`, `b=2`, `c=4`, and `d=8`, then any combination of the group has a unique sum that can be stored in an integer. The pair `a` and `c` would be 5 (binary 0101), while `b`, `c`, and `d` would be 14 (binary 1110). In wxPython, the attributes have symbolic constants, so you don't need to worry about the individual bit values.

Styles are defined for all wxPython widgets by passing a bitmask to the style parameter of the constructor. Some widgets also define a `SetStyle()` method, allowing you to change the style after the widget is created. All the individual style elements that you might use have a predefined constant identifier (such as `wx.MINIMIZE_BOX`). To add multiple styles together, you use the Python bitwise OR operator, `|`. For example, the constant `wx.DEFAULT_FRAME_STYLE` is defined as a combination of basic style elements:

```
wx.MAXIMIZE_BOX | wx.MINIMIZE_BOX | wx.RESIZE_BORDER |
wx.SYSTEM_MENU | wx.CAPTION | wx.CLOSE_BOX
```

To remove individual style bits from a composed style, you use the bitwise exclusive or (XOR) operator, `^`. For example, to create a window that is based on the default, but which is not resizable by the user, you could do this:

```
wx.DEFAULT_FRAME_STYLE ^ (wx.RESIZE_BORDER | wx.MINIMIZE_BOX |
wx.MAXIMIZE_BOX)
```

It is highly recommended that you use the default style on top-level frames so that the user can easily recognize them as being top level. At the very least, you need to ensure that there is some way for a top-level frame to be closed. This is most easily done by including the `wx.SYSTEM_MENU` style. Also be aware that by inadvertently using a bitwise AND (`&`) operation, instead of a bitwise OR, you can easily wind up with no styles chosen, resulting in an unbordered frame that you cannot move, resize, or close. This is, of course, not recommended.

Table 2.2 has a listing of the most important styles for `wx.Frame`.

Table 2.2 Some of the most commonly used style parameters for `wx.Frame`

Style	Description
<code>wx.CAPTION</code>	Adds a title bar on the frame, which displays the frame's <code>Title</code> property.
<code>wx.CLOSE_BOX</code>	Instructs the system to display a close box on the frame's title bar, using the system defaults for placement and style. Also enables the close item on the system menu if applicable.

continued on next page

Table 2.2 Some of the most commonly used style parameters for `wx.Frame` (continued)

Style	Description
<code>wx.DEFAULT_FRAME_STYLE</code>	As you might expect from the name, this is the default if no style is specified. It is defined as <code>wx.MAXIMIZE_BOX wx.MINIMIZE_BOX wx.RESIZE_BORDER wx.SYSTEM_MENU wx.CAPTION wx.CLOSE_BOX</code> .
<code>wx.FRAME_SHAPED</code>	Frames created with this style can use the <code>SetShape()</code> method to create a window with a non-rectangular shape.
<code>wx.FRAME_TOOL_WINDOW</code>	Makes the frame look like a toolbox window by giving it a smaller titlebar than normal. Under Windows a frame created with this style does not show in the taskbar listing of all open windows.
<code>wx.MAXIMIZE_BOX</code>	Adds a maximize box on the frame, using the system parameters for the look and placement of the box. Also enables maximize functionality in the system menu if applicable.
<code>wx.MINIMIZE_BOX</code>	Adds a minimize box on the frame, using the system parameters for the look and placement of the box. Also enables minimize functionality in the system menu if applicable.
<code>wx.RESIZE_BORDER</code>	Adds a resizable border to the frame.
<code>wx.SIMPLE_BORDER</code>	A plain border without decoration. May not work on all platforms.
<code>wx.SYSTEM_MENU</code>	Adds the system menu (with close, move, resize, etc. functionality, using system look and feel) and the close box to the window. The availability of resize and close operations within this menu depends on the styles <code>wx.MAXIMIZE_BOX</code> , <code>wx.MINIMIZE_BOX</code> and <code>wx.CLOSE_BOX</code> being chosen.

The next four figures show a few common frame styles.

Figure 2.4 was created with `wx.DEFAULT_STYLE`. Figure 2.5 is a frame created using the non-resizable style combination shown in the previous code snippet.



Figure 2.4 A frame created with the default style

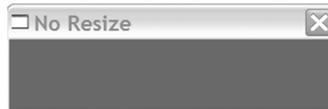


Figure 2.5 A frame created to be non-resizable. Notice the lack of minimize/maximize buttons.

Figure 2.6 uses `style=wx.DEFAULT_FRAME_STYLE | wx.FRAME_TOOL_WINDOW`.



Figure 2.6 A toolbar frame, with a smaller title bar and no system menu



Figure 2.7 A frame with a help button

Figure 2.7 uses the extended style `wx.help.FRAME_EX_CONTEXTHELP`, which is described in chapter 8.

Now that we've seen how to create `wx.Frame` objects, we'll start to show how to make them useful, by adding additional widgets inside the frame.

2.6 How do I add objects and subwindows to a frame?

We've described how to create `wx.Frame` objects, but as yet the frames are not very interesting. For one thing, they are empty. In this section, we'll show you the basics of inserting objects and subwindows inside your frame for the user to interact with.

2.6.1 Adding widgets to a frame

Figure 2.8 shows a custom frame subclass called `InsertFrame`. When the close button is clicked, the window will close and the application will end.

Listing 2.3 defines the `wx.Frame` subclass shown in figure 2.8. Not all of the concepts in this snippet have been covered yet, so don't worry if some things are not clear.



Figure 2.8 The `InsertFrame` window is an example demonstrating the basics of inserting items into a frame.

Listing 2.3 The `InsertFrame` code

```
#!/usr/bin/env python
import wx

class InsertFrame(wx.Frame):
    def __init__(self, parent, id):
        wx.Frame.__init__(self, parent, id, 'Frame With Button',
                           size=(300, 100))
        panel = wx.Panel(self)
        button = wx.Button(panel, label="Close", pos=(125, 10),
```

Adding the button
to the panel

2

1 Creating the panel

←

```

        size=(50, 50)
        self.Bind(wx.EVT_BUTTON, self.OnCloseMe, button)
        self.Bind(wx.EVT_CLOSE, self.OnCloseWindow)
    def OnCloseMe(self, event):
        self.Close(True)
    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = InsertFrame(parent=None, id=-1)
    frame.Show()
    app.MainLoop()

```

3 Binding the button click event

Binding the window close event

4

2

1

The `__init__` method of the `InsertFrame` class creates two subwindows **1**, **2**. The first is a `wx.Panel`, which is essentially a plain container for other windows and has little functionality of its own. The second is a `wx.Button`, which is an ordinary button. Next, the button click event and the close window event are bound to the function that will be executed when the event takes place **3**, **4**.

In most cases, you will create a single `wx.Panel` instance the same size as your `wx.Frame` to hold all of the contents of your frame. Doing so keeps the custom contents of the window separate from other elements such as the toolbar and status bar. In addition, on Windows operating systems, the default background color of a `wx.Frame` is not standard (it's gray, not white), while a `wx.Panel` will have a white background by default (assuming you haven't changed your system's color and theme settings). The `wx.Panel` class also enables traversal of the elements inside via the tab button, which `wx.Frame` does not.

If you are familiar with other UI toolkits, it may seem strange that you do not need to explicitly call an add method to insert a subwindow into a parent. Instead, in wxPython you just specify the parent window when the subwindow is created, and the subwindow is implicitly added inside that parent object, as is done in listing 2.3 **2**.

You might also wonder why the `wx.Button` in listing 2.5 is created with an explicit position and size, while the `wx.Panel` is not. In wxPython, if a frame is created with just a single child window, then that child window (in this case, the `wx.Panel`) is automatically resized to fill the client area of the frame. This automatic resizing will override any position or size information for the child—even if a position or size had been specified for the panel, it would have been ignored. This automatic resizing only happens for a single element if it is within frames or

dialogs. The button is a child of the panel, not the frame, so its specified size and position are used. If a size and position had not been specified for the button, it would have been placed in the default position, which is the upper left corner of the panel, with its size based on the length of the label.

Explicitly specifying the size and position of every subwindow can get tedious quickly. More importantly, it doesn't allow your application to reposition objects gracefully when the user resizes a window. To solve both of these problems, wxPython uses objects called *sizers* to manage complex placement of child windows. Sizers will be covered briefly in chapter 7 and in more detail in part 2.

2.6.2 Adding a menubar, toolbar, or status bar to a frame

Often, an application window will have one or more of three special subwindows—a menubar at the top, a toolbar below that, and a status bar at the bottom. This is so common that wxPython provides special shortcut methods for the creation of toolbars and status bars (and menus aren't that hard either). Figure 2.9 shows a sample frame with a menubar, toolbar, and status bar.

Listing 2.4 shows the `__init__` method which decorates a plain window with all three subwindows. Again, this is just an overview, and the classes in question will be covered in more detail in chapter 7 and later in part 2.

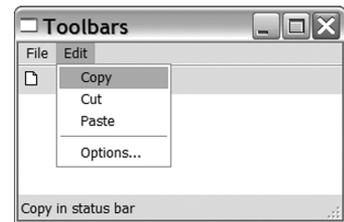


Figure 2.9 A sample frame with menubar, toolbar, and status bar

Listing 2.4 Creating toolbars

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

```
import wx
import images
```

```
class ToolbarFrame(wx.Frame):
```

```
    def __init__(self, parent, id):
```

```
        wx.Frame.__init__(self, parent, id, 'Toolbars',
                           size=(300, 200))
```

```
        panel = wx.Panel(self)
```

```
        panel.SetBackgroundColour('White')
```

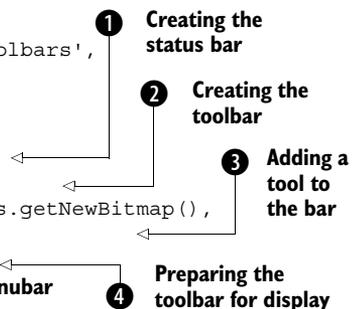
```
        statusBar = self.CreateStatusBar()
```

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

```
        toolbar.AddSimpleTool(wx.NewId(), images.getNewBitmap(),
                               "New", "Long help for 'New'")
```

```
        toolbar.Realize()
```

```
        menuBar = wx.MenuBar()
```



```

menu1 = wx.Menu()
menuBar.Append(menu1, "&File")
menu2 = wx.Menu()
menu2.Append(wx.NewId(), "&Copy", "Copy in status bar")
menu2.Append(wx.NewId(), "C&ut", "")
menu2.Append(wx.NewId(), "Paste", "")
menu2.AppendSeparator()
menu2.Append(wx.NewId(), "&Options...", "Display Options")
menuBar.Append(menu2, "&Edit")
self.SetMenuBar(menuBar)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = ToolbarFrame(parent=None, id=-1)
    frame.Show()
    app.MainLoop()

```

5 Creating two individual menus

6 Creating individual menu items

Attaching the menu to the menubar

Attaching the menubar to the frame

- ❶ This line creates a status bar, which is an instance of the class `wx.StatusBar`. As far as the frame is concerned, it's a subwindow placed at the bottom of the frame whose width is the same as the width of the frame, and whose height is determined by the underlying operating system. The purpose of the status bar is to display text set by various events in the application. The text size and font of the status window are also set by the underlying system.
- ❷ Creates an instance of `wx.ToolBar`, which is a container of command buttons automatically positioned at the top of the frame.
- ❸ There are two methods to add tools to your toolbar; this line shows the one with fewer arguments, `AddSimpleTool()`. The arguments are an ID, a bitmap, short help text to display as a tooltip for the item, and longer help text to be displayed in the status bar for the item. (Don't worry about where the bitmap is coming from at the moment.) Although the tool as displayed on the toolbar will look like a button, in terms of the event system, it behaves more like a menu item, which makes it easy for a tool to act as an exact duplicate for an item in your menubar. Specifically, pressing the tool button triggers the same kind of wxPython event as the menu item, meaning that both can be responded to by the same method.
- ❹ The `Realize()` method tells the toolbar to calculate where the buttons should be positioned, and is required if the toolbar has any tools added that depend on the toolbar's dynamic positioning because they do not specify their own size or position explicitly.
- ❺ Creates a `wx.Menu` object, which represents a single menu on the bar. (We created two of them, but only so that we could take a screen shot of the pull-down without covering the toolbar icon.)

- 6 Creates the individual menu items, with the arguments representing an ID, the text of the item, and the text displayed on the status bar when the menu is moused over. The “&” indicates the character to be chosen for a menu accelerator.

In addition to using widgets in your frames, you’ll also communicate with users via standard dialogs, which is made easy using wxPython’s predefined dialog functions.

2.7 How can I use common dialogs?

The wxPython library provides a rich set of predefined dialog boxes that you can use to give your application a consistent, familiar look and feel. In this section, we’ll discuss three basic ways to get information from a user in a dialog:

- 1 Message dialog
- 2 Text entry
- 3 Choosing from a list

There are many other standard dialogs in wxPython, including a file picker, a color picker, progress dialog, print setup, and a font picker. These will be discussed in chapter 9.

Sending a message without using Western Union

The most basic mechanism for communicating with the user is `wx.MessageDialog`, which is a simple alert box. The `wx.MessageDialog` can be used as both a simple OK box, or as a yes/no dialog. The following snippet shows the yes/no behavior:

```
dlg = wx.MessageDialog(None, 'Is this the coolest thing ever!',
                       'MessageDialog', wx.YES_NO | wx.ICON_QUESTION)
result = dlg.ShowModal()
dlg.Destroy()
```

The resulting dialog looks like figure 2.10, and the signature for the constructor is:

```
wx.MessageDialog(parent, message,
                 caption="Message box",
                 style=wx.OK | wx.CANCEL,
                 pos=wx.DefaultPosition)
```

where `parent` is a parent window, or `None` if the dialog should be top-level. The `message` is the string which appears in the dialog, and the `caption` is the string that appears in the dialog’s title bar. The `style` parameter is



Figure 2.10 A message dialog, configured for a yes/no response

a bitmask which covers the buttons and icons displayed in the dialog. The `pos` parameter takes either a `wx.Point` or Python tuple, and allows you to specify the position of the dialog on the display, if you so desire.

The `ShowModal()` method displays the dialog as a modal frame, meaning that no other window in the application will respond to user events until the dialog is closed. This method is common to all dialogs. The return value of `ShowModal()` is an integer, signifying the result of the dialog. In this case, the result corresponds to the button pressed by the user. For a `wx.MessageDialog`, it will be one of the following constants: `wx.ID_YES`, `wx.ID_NO`, `wx.ID_CANCEL`, or `wx.ID_OK`.

The style bits can be combined to manage the buttons displayed, with the legal values being `wx.OK`, `wx.CANCEL`, or `wx.YES_NO`. Style bits can also be used to set the icon displayed on the window, which is one of `wx.ICON_ERROR`, `wx.ICON_EXCLAMATION`, `wx.ICON_INFORMATION`, and `wx.ICON_QUESTION`.

Just a single line of text, please

If you need to get a single line of text from the user, then you can use the class `wx.TextEntryDialog`, as follows. This snippet creates the dialog, and retrieves the entered value if the user exits with a click on the OK button:

```
dlg = wx.TextEntryDialog(None, "Who is buried in Grant's tomb?",
                        'A Question', 'Cary Grant')
if dlg.ShowModal() == wx.ID_OK:
    response = dlg.GetValue()
```

Figure 2.11 shows what the dialog looks like in use.

The parameters of the `wx.TextEntryDialog` constructor are, in order, a parent window, the text caption inside the window, the text caption for the outside of the window (which defaults to “Please enter text”), and a default value for the user entry field (which defaults to the empty string). There is also a `style` argument which defaults to `wx.OK | wx.CANCEL`. You can use the `style` attribute to display the dialog without a cancel box, by passing only the value `wx.OK`. As with the `wx.MessageDialog`, the `ShowModal()` method returns the ID of the button pressed. The `GetValue()` method contains the value the user entered into the text field (there’s a parallel `SetValue()` method that allows you to change the field programmatically).

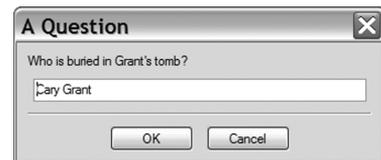


Figure 2.11 A text entry dialog

Choose from the following list

If, instead of all the freedom that comes with being able to enter any thing that enters his head, you want the user to only be able to select a choice from a provided list, you can use the `wx.SingleChoiceDialog` class. A simple usage is shown here:

```
dlg = wx.SingleChoiceDialog(None,
    'What version of Python are you using?',
    'Single Choice',
    ['1.5.2', '2.0', '2.1.3', '2.2', '2.3.1'],
    if dlg.ShowModal() == wx.ID_OK:
        response = dlg.GetStringSelection()
```

Figure 2.12 shows the resulting dialog box. The parameters are similar to the text entry dialog, except that instead of passing a string default text, you pass a list of strings that are displayed in the list. There are two ways that you can get the returned selection. The method `GetSelection()` returns the index of the user selection, while `GetStringSelection()` returns the actual string selected.

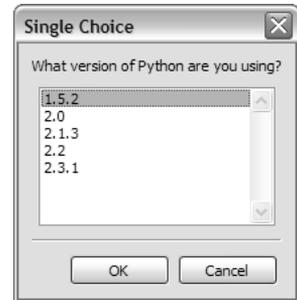


Figure 2.12 The `SingleChoiceDialog` window, allowing a user to choose from a predefined list

2.8 What are some common errors with application objects and frames?

There are a few errors that can happen in the creation of your wxPython application object or initial top-level window that can be difficult to track down, especially if you are seeing the error for the first time. The errors listed here are the kind of thing where the error message is not necessarily completely descriptive of the actual problem, or where the error can be difficult to diagnose the first time you see it. These are all fairly common errors for new wxPython programmers.

Here is a troubleshooting guide for some of the most common symptoms:

Symptom Error message at startup saying “unable to import module wx.”

Cause—The wxPython module is not in your `PYTHONPATH`. This means wxPython may not have been correctly installed. Alternately, if there is more than one version of Python on the system, wxPython may have been installed against a different one than the one you are using.

Solution—Determine which Python version or versions are installed on the machine you are using. On a Unix system, the command `which python` should tell you the default installation. On a Windows system, you may have to go into the folder options dialog and see what application `.py` files are assigned to. If wxPython is correctly installed for that Python version, it puts itself in the `<python-home>/Lib/site-packages` subdirectory. You will likely need to install or reinstall wxPython.

Symptom *The application crashes immediately on startup, or a blank window is displayed, followed immediately by an application crash.*

Cause—A wxPython object is created or used before the creation of the `wx.App`.

Solution—Create the `wx.App` object immediately on starting your script.

Symptom *My top-level windows are created and immediately close. The application exits immediately.*

Cause—The method `wx.App.MainLoop()` was not called.

Solution—Call the `MainLoop()` method after all your setup is complete.

Symptom *My top-level windows are created and immediately close. The application exits immediately, and I did call `MainLoop()`.*

Cause—An error in your application's `OnInit()` method, or some method called by it (such as a frame `__init__()` method).

Solution—An error before `MainLoop()` is called can trigger an exception that exits the program. If your application is set to redirect `stdout` and `stderr` to windows, then those windows will not display long enough for you to see the error. Create your application object with the `redirect=False` option to allow you to see the error message. See the section “Redirecting Output” for more details.

2.9 Summary

- The foundation of a wxPython program is based on two required objects: an application object and the top-level window. Every wxPython application needs to instantiate one instance of `wx.App`, and have at least one top-level window.

- The application object contains the `wx.App.OnInit()` method, which is called on startup. This method is the preferred place to initialize frames and other global objects. A wxPython application normally shuts down when all of its top-level windows have closed, or when the main event loop is otherwise exited.
- The application object also controls where wxPython directs textual output. By default, wxPython redirects `stdout` and `stderr` to a special window. This behavior can make it hard to troubleshoot startup errors. Luckily, it's no trouble at all to change this behavior to allow wxPython to send error messages to a file or console window.
- A wxPython application usually has at least one subclass of `wx.Frame`. A `wx.Frame` object can be created in multiple styles using the `style` parameter. Every `wx.Widgets` object, including frames, has an ID, which can be explicitly assigned by the application or generated by wxPython. Subwindows are the meat of a frame, inserted into a frame by creating the subwindow with that frame as a parent. Usually, a frame contains a single `wx.Panel` and further subwindows are placed in the panel. A frame's single subwindow is automatically resized when the parent frame resizes. Frames have explicit mechanisms for managing a menubar, toolbar, and status bar.
- Although you'll use frames for anything complex, when you want to simply get quick information from a user, you can show the user a standard dialog window. There are standard dialogs for many tasks, including an alert box, simple text entry, and entry from a list.

Now that we've talked about the foundation of a wxPython program in terms of the required data objects, we'll start to talk about the basic blocks of a wxPython program as it manages the flow of control while running. In the next chapter, we'll talk about events and the event loop.

wxPython IN ACTION

Noel Rappin ■ Robin Dunn

If you add the powerful wxWidgets toolkit to Python, you get wxPython: an open source GUI framework with a well-deserved reputation for simplicity and ease of use. wxPython lets you build cross-platform applications that have robust, highly functional graphical user interfaces.

The first book on the subject, **wxPython in Action** offers a friendly tutorial to get you started, a detailed guide to best practices, and an extensive reference for wxPython's large widget library. It covers an impressive amount of information delivered at a measured pace, encouraging experimentation and learning by doing.

The book's direct, no-nonsense style makes for an easy introduction to the concepts. It offers a complete discussion of when, why, and how to use the many widgets in the toolkit. And it includes dozens of handy reference tables so you can easily look up object properties, methods, and events. Co-authored by wxPython creator Robin Dunn, **wxPython in Action** is the authoritative book on the subject.

What's Inside

- Create professional GUIs with wxPython
- Program in an event-oriented framework
- Use wxPython sizers for your layout
- Refactor and unit test to improve your programs
- A reference to wxPython's powerful widget set

Noel Rappin is a senior software engineer at Motorola and a leading Python practitioner. He has a Ph.D. from the Georgia Institute of Technology and lives in Chicago, IL.

A veteran of the software industry for almost two decades, **Robin Dunn** is the creator and maintainer of wxPython. Geographically, he is to be found in or between Portland, OR and Vancouver, WA.

"Clear and to the point"

—Kevin Ollivier
Software Developer
Tulane University

"... I had many 'aha! so that's what that is for' moments—this book is a winner!"

—Dave Brueck, Chief Architect
Move Networks

"Excellent code examples"

—Dr. Pim Van Heuven
Technical Director, Think-Wize

"Extremely high technical quality"

—Chris Mellon
Independent Consultant

"Outstanding!"

—Doug Tillman
Software Developer

"... I recommend it, both as an introduction and as a reference."

—Dr. Stefan Neis
Software Developer, KOBIL



Ask the Authors



Ebook edition

www.manning.com/rappin



9 781932 394627

ISBN 1-932394-62-1