

SAMPLE CHAPTER



wxPython

IN ACTION

Noel Rappin
Robin Dunn

 MANNING



wxPython in Action

by Noel Rappin

and

Robin Dunn

Sample Chapter 9

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 users choices with dialogs

This chapter covers

- Creating modal dialogs and alert boxes
- Using standard dialogs
- Creating wizards
- Showing startup tips
- Creating validators and using them to transfer data

Where frames are used for long-term interactions with the user, a dialog is typically used to get a small amount of information from the user, and is then quickly dispatched. Dialog windows are often *modal*, which means that no other frame in the application can handle events until the dialog is closed. In this chapter we will discuss the many varieties of dialogs available in wxPython. In addition to allowing you to create your own dialog styles, wxPython provides you with several predefined dialog types. These predefined dialogs include both simple interactions, such as a basic alert box, and more complex dialogs that mimic system interactions, such as page layout or file selection.

9.1 Working with modal dialogs

Modal dialogs are used for quick interactions with the user or for any time that information in a dialog absolutely must be entered before the user can move forward in the program. Within wxPython, there are several standard functions to display basic modal dialogs. These dialogs include alert boxes, one line of text entry, and choosing from a list. In the following sections, we'll show you what these dialogs look like, and how using the predefined functions will make your life easier.

9.1.1 How do I create a modal dialog?

A *modal dialog* blocks other widgets from receiving user events until it is closed; in other words, it places the user in dialog mode for the duration of its existence. As you can see from figure 9.1, you can't always distinguish between dialogs and frames by their appearance. In wxPython, the difference between a dialog and a frame is not based on how they display, but is largely a matter of the way in which they handle events.

A dialog is created and deployed somewhat differently from a frame. Listing 9.1 shows the code used to generate figure 9.1. After a dialog is displayed and a button is clicked, the dialog closes, and a message is printed to `stdout`.

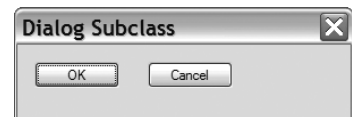


Figure 9.1
A sample modal dialog

Listing 9.1 Defining a modal dialog

```
import wx

class SubclassDialog(wx.Dialog):
    def __init__(self):
        wx.Dialog.__init__(self, None, -1, 'Dialog Subclass',
                           size=(300, 100))
```

← Initializing the dialog

```
okButton = wx.Button(self, wx.ID_OK, "OK", pos=(15, 15))
okButton.SetDefault()
cancelButton = wx.Button(self, wx.ID_CANCEL, "Cancel",
                          pos=(115, 15))

if __name__ == '__main__':
    app = wx.PySimpleApp()
    dialog = SubclassDialog()
    result = dialog.ShowModal()  ← Showing the modal dialog
    if result == wx.ID_OK:
        print "OK"
    else:
        print "Cancel"
    dialog.Destroy()
```

Compared to the `wx.Frame` examples in the previous chapter, there are a couple of interesting things to note about this code. In the `__init__` method, the button is added directly to `wx.Dialog`, rather than to a `wx.Panel`. Panels are used much less commonly in dialogs than in frames, partially because dialogs tend to be simpler than frames, but largely because the features of a `wx.Panel` (standard system background and tab key transversal through the controls) already exist by default in `wx.Dialog`.

To get the dialog to display modally, use the `ShowModal()` method. This has a different effect on program execution than the modeless `Show()` method used for frames. Your application will wait at the point of the `ShowModal()` call until the dialog is dismissed. The dialog being shown is the only part of the wxPython application that receives user events during that time, although system windows from other applications will still work.

The mode continues until the dialog method `EndModal(retCode)` is called, which closes the dialog. The `retCode` argument is an integer value, which is then also returned by the original `ShowModal()` method. Typically, the application uses the return value to learn how the user closed the dialog as a guide to future processing. However, ending the mode does not destroy or even close the dialog. Keeping the dialog around can be a good thing, because it means that you can store information about the user's selections as data members of the dialog instance, and recover that information from the dialog even after the dialog is closed. In the next sections, we'll see examples of that pattern as we deal with dialogs where the user enters data for use elsewhere in the program.

Since there are no event handlers defined in listing 9.1, you may be wondering how the dialog does anything in response to the button clicks. The behavior is already defined in `wx.Dialog`. There are two predefined wxPython ID numbers

that have special meaning in dialogs. When a `wx.Button` with the ID `wx.ID_OK` is clicked in a dialog, the mode is ended, the dialog is closed, and `wx.ID_OK` is the return value of the `ShowModal()` call. Similarly, a button with the ID `wx.ID_CANCEL` does the same things, but returns the value `wx.ID_CANCEL`. It's up to the rest of the application to ensure that the semantics of OK and Cancel are appropriately enforced.

Listing 9.1 displays a typical method of dealing with a modal dialog. After the dialog is invoked, the return value is used as the test in an `if` statement. In this case, we simply print the result. In a more complex example, the `wx.ID_OK` branch would implement the actions that the user took within the dialog, such as opening the file or choosing the color.

Typically you should explicitly destroy a dialog when you are finished with it. This signals the C++ object that it should destroy itself which will then allow the Python parts of it to be garbage collected. If you wish to reuse the dialog later in your application without having to recreate it, perhaps to speed the response time for complex dialogs, you can keep a reference to the dialog and simply call its `ShowModal()` method when you need to activate it again. Be sure to destroy it when the application is ready to exit, otherwise `MainLoop()` will see it as a still existing top-level window and will not exit normally.

9.1.2 How do I create an alert box?

The three simplest ways of interacting with the user via a dialog box are `wx.MessageDialog`, which represents an alert box, `wx.TextEntryDialog`, which prompts the user to enter some short text, and `wx.SingleChoiceDialog`, which allows the user to select from a list of available options. The next three sections discuss these simple dialogs.

A message box dialog displays a short message and allows the user to press a button in response. Typically, message boxes are used to display important alerts, yes/no questions, or to ask the user to continue with or cancel some action. Figure 9.2 displays a typical message box.

Using a message box is quite simple. Listing 9.2 displays two ways of creating a message box.

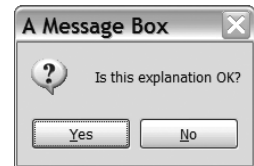


Figure 9.2 A standard message box, in a yes/no configuration

Listing 9.2 Creating a message box

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
```

```

dlg = wx.MessageDialog(None, "Is this explanation OK?",
                       'A Message Box',
                       wx.YES_NO | wx.ICON_QUESTION)
retCode = dlg.ShowModal()
if (retCode == wx.ID_YES):
    print "yes"
else:
    print "no"
dlg.Destroy()

retCode = wx.MessageBox("Is this way easier?", "Via Function",
                       wx.YES_NO | wx.ICON_QUESTION)

```

Using
a class

Using a function ①

Listing 9.2 creates two message boxes, one after the other. The first method creates an instance of the class `wx.MessageDialog`, and displays it using `ShowModal()`.

Using the `wx.MessageDialog` class

Using the constructor for the `wx.MessageDialog`, you can set the message and buttons for the dialog, as in the following:

```

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

```

The message argument is the text that is actually displayed inside the body of the dialog. If the message string contains `\n` newline characters, there will be line breaks in the text. The `caption` argument is displayed in the title box of the dialog. The `pos` argument allows you to specify where the dialog is displayed on the screen—under MS Windows, this argument is ignored.

The style flags for a `wx.MessageDialog` split into two types. The first type controls the buttons that display in the dialog. Table 9.1 describes these styles.

Table 9.1 Button styles for a `wx.MessageDialog`

Button Style	Description
<code>wx.CANCEL</code>	Include a cancel button. This button will have the ID value of <code>wx.ID_CANCEL</code> .
<code>wx.NO_DEFAULT</code>	In a <code>wx.YES_NO</code> dialog, the No button is the default.
<code>wx.OK</code>	Include an OK button. This button will have the ID value of <code>wx.ID_OK</code> .
<code>wx.YES_DEFAULT</code>	In a <code>wx.YES_NO</code> dialog, the Yes button is the default. This is the default behavior.
<code>wx.YES_NO</code>	Include buttons labeled Yes and No, with the ID values of <code>wx.ID_YES</code> and <code>wx.ID_NO</code> , respectively.

The second set of style flags controls the icon displayed next to the message text. Those styles are listed in Table 9.2.

Table 9.2 Icon styles for a `wx.MessageDialog`

Style	Description
<code>wx.ICON_ERROR</code>	An icon indicating an error.
<code>wx.ICON_EXCLAMATION</code>	An icon indicating an alert.
<code>wx.ICON_HAND</code>	The same as <code>wx.ICON_ERROR</code>
<code>wx.ICON_INFORMATION</code>	The letter "i" information icon.
<code>wx.ICON_QUESTION</code>	A question mark icon.

Finally, you can use the style `wx.STAY_ON_TOP` to display the dialog above any other windows in the system, including system windows and wxPython application windows.

As you can see in listing 9.2, the dialog is invoked using `ShowModal()`. Depending on the displayed buttons, the result is either `wx.ID_OK`, `wx.ID_CANCEL`, `wx.ID_YES`, or `wx.ID_NO`. As with other dialogs, you'll typically use the response value to control program execution in response to the dialog.

Using the `wx.MessageBox()` function

Line ❶ of listing 9.2 displays a shorter method for invoking a message box. The convenience function `wx.MessageBox()` creates the dialog, calls `ShowModal()`, and returns, `wx.YES`, `wx.NO`, `wx.CANCEL`, or `wx.OK`. The signature of the function is simpler than the constructor for the `MessageDialog` object, as in:

```
wx.MessageBox(message, caption="Message", style=wx.OK)
```

In this example, `message`, `caption`, and `style` have the same meanings as in the constructor, and you can use all of the same style flags. As we'll see throughout this chapter, several of the predefined dialogs in wxPython also have convenience functions. As long as you are creating the dialogs for a single use, the mechanism you choose is a matter of preference. If you plan to hold onto the dialog to invoke it more than once, it may be preferable to instantiate yourself the object so you can hold onto the reference, although for simple dialogs such as these, the time saved is probably negligible.

To display a lot of text in your message box (i.e., an end-user license agreement display), you can use the wxPython-specific class `wx.lib.dialogs.ScrolledMessageDialog`, which contains the following constructor:

```
wx.lib.dialogs.ScrolledMessageDialog(parent, msg, caption,
                                     pos=wx.wxDefaultPosition, size=(500,300))
```

This dialog doesn't use the native message box widget, it builds a dialog from other wxPython widgets. It only displays an OK button, and takes no further style information.

9.1.3 How do I get short text from the user?

The second simple type of dialog box is `wx.TextEntryDialog`, which is used to get short text entry from the user. Typically, you'll see this used when requesting a username or password at the beginning of a program, or as a very rudimentary replacement for a data entry form. Figure 9.3 displays a typical text dialog.

The code for this example is displayed in listing 9.3.

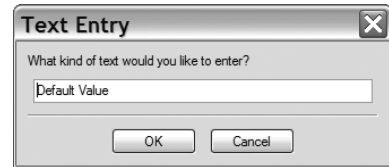


Figure 9.3
A text entry standard dialog

Listing 9.3 Code for text entry

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = wx.TextEntryDialog(None,
                               "What kind of text would you like to enter?",
                               "Text Entry", "Default Value", style=wx.OK|wx.CANCEL)
    if dialog.ShowModal() == wx.ID_OK:
        print "You entered: %s" % dialog.GetValue()
    dialog.Destroy()
```

As in the previous section, we create an instance of a dialog class, in this case `wx.TextEntryDialog`. The constructor for this class is a bit more complex than the simple message dialog:

```
wx.TextEntryDialog(parent, message, caption="Please enter text",
                  defaultValue="", style=wx.OK | wx.CANCEL | wx.CENTRE,
                  pos=wx.DefaultPosition)
```

The message argument is the text prompt that is displayed in the dialog, while the caption is displayed in the title bar. The `defaultValue`, if set, is displayed inside the text box. The style information can include `wx.OK` and `wx.CANCEL`, which displays the appropriate button.

Several of the styles from an ordinary `wx.TextCtrl` can also be set here. The most useful would be `wx.TE_PASSWORD`, which masks the input for securely entering a password. You can also use `wx.TE_MULTILINE` to allow the user to enter more than one line of text in the dialog, and `wx.TE_LEFT`, `wx.TE_CENTRE`, and `wx.TE_RIGHT` to adjust the justification of the entered text.

The last line of listing 9.3 displays another difference between the text box and the message box. The information entered by the user is stored in the dialog instance, and must be retrieved by the application afterwards. In this case, you can get at the value by using the dialog's `GetValue()` method. Remember, if the user presses `Cancel` to exit the dialog, it means they don't want you to use his entered value. You can also programmatically set the value with the `SetValue()` method.

The following are convenience functions for dealing with text dialogs:

- `wx.GetTextFromUser()`
- `wx.GetPasswordFromUser()`
- `wx.GetNumberFromUser()`

Most similar to the usage in listing 9.3 is `wx.GetTextFromUser`:

```
wx.GetTextFromUser(message, caption="Input text",
                  default_value="", parent=None)
```

In this example, `message`, `caption`, `default_value`, and `parent` are all in the `wx.TextEntryDialog` constructor. If the user presses `OK`, the return value of the function is the user entered string. If the user presses `Cancel`, the function returns the empty string.

If you want the user to enter a masked password, you can use the `wx.GetPasswordFromUser` function.

```
wx.GetPasswordFromUser(message, caption="Input text",
                       default_value="", parent=None)
```

In this example, the arguments mean what you'd expect. The user input is displayed as asterisks, and the return value is as in the previous function—the string if the user hits `OK`, an empty string if the user hits `cancel`.

Finally, you can request a number from a user with the `wx.GetNumberFromUserMethod`.

```
wx.GetNumberFromUser(message, prompt, caption, value, min=0,
                    max=100, parent=None)
```

The argument names here are a bit different. The `message` is an arbitrarily long message displayed above the prompt string, which is directly above the text entry field. The `value` argument is a numeric `long`, and is the default value. The `min` and `max` arguments allow you to specify a valid range for user input. If the user exits with the OK button, the method returns the entered value, converted to a `long`. If the value cannot be converted to a number, or the value is outside the `min` and `max` range, the function returns `-1`, which means that if you use this function for a range of negative numbers, you may want to consider an alternate plan.

9.1.4 How can I display a list of choices in a dialog?

If allowing your users a blank text entry seems like too much freedom, you can restrict their options by using `wx.SingleChoiceDialog` to give them a single choice out of a group of options. Figure 9.4 displays an example.

The essential code displayed in listing 9.4 is similar to the dialog examples we've already discussed in this chapter.

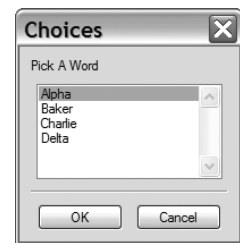


Figure 9.4 A single choice dialog

Listing 9.4 Displaying a dialog list of choices

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    choices = ["Alpha", "Baker", "Charlie", "Delta"]
    dialog = wx.SingleChoiceDialog(None, "Pick A Word", "Choices",
                                   choices)
    if dialog.ShowModal() == wx.ID_OK:
        print "You selected: %s\n" % dialog.GetStringSelection()
    dialog.Destroy()
```

The constructor for the `wx.SingleChoiceDialog` is as follows:

```
wx.SingleChoiceDialog(parent, message, caption, choices,
                      clientData=None, style=wx.OK | wx.CANCEL | wx.CENTRE,
                      pos=wx.DefaultPosition)
```

The `message` and `caption` arguments are as before, displaying the prompt in the dialog and the title bar, respectively. The `choices` argument takes a Python list of strings, and they are, as you might suspect, the choices presented in the dialog. The `style` argument has the three options that are in the default, allowing an OK button, a Cancel button, and the option to center the dialog on the screen. The

centre option does not work on Windows operating systems, and neither does the `pos` argument.

If you want to set the dialog default before the user sees it, use the method `SetSelection(selection)`. The argument to that method is the integer index of the selection, and not the actual string to be selected. After the user has made a selection, you can retrieve it by using either `GetSelection()`, which returns the integer index of the selected option, or `GetStringSelection()` which returns the actual selected string.

There are two convenience functions for single choice dialogs. The first, `wx.GetSingleChoice`, returns the string that the user selected.

```
wx.GetSingleChoice(message, caption, aChoices, parent=None)
```

The `message`, `caption`, and `parent` arguments are as in the `wx.SingleChoiceDialog` constructor. The `aChoices` argument is the list of items. The return value is the selected string if the user presses OK, and the empty string if the user presses Cancel. This means that if the empty string is a valid choice, you should probably not use this function.

Instead, you might use `wx.GetSingleChoiceIndex`.

```
wx.GetSingleChoiceIndex(message, caption, aChoices, parent=None)
```

This function has the same arguments, but a different return value. It returns the index of the user choice if OK, and `-1` if the user hits Cancel.

9.1.5 How can I display progress?

In many programs, the program needs to go off and do something by itself unencumbered by user input. At that time, it's customary for the program to give the user some visual indication that it's actually doing something. In wxPython, that is often managed with a progress box, as displayed in figure 9.5.

The sample code to generate this progress box is displayed in listing 9.5.

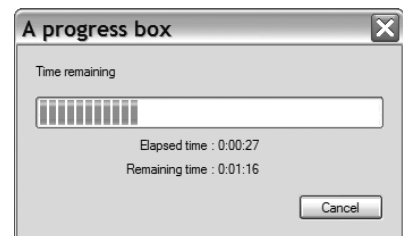


Figure 9.5 A sample progress box, joined in progress

Listing 9.5 Generating a sample progress box

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
```

```

progressMax = 100
dialog = wx.ProgressDialog("A progress box",
    "Time remaining", progressMax,
    style=wx.PD_CAN_ABORT | wx.PD_ELAPSED_TIME |
    wx.PD_REMAINING_TIME)
keepGoing = True
count = 0
while keepGoing and count < progressMax:
    count = count + 1
    wx.Sleep(1)
    keepGoing = dialog.Update(count)
dialog.Destroy()

```

All the options for the progress box are set in the constructor, which looks like this:

```

wx.ProgressDialog(title, message, maximum=100, parent=None,
    style=wx.PD_AUTO_HIDE | wx.PD_APP_MODAL)

```

The arguments are different than in other dialog boxes. The title is placed in the title bar of the window, and the message is displayed in the dialog itself. The maximum is the highest possible value of the counter you are using to display progress. As you can tell from figure 9.5, the user does not see this number, so feel free to make it any value that is convenient for your application.

Table 9.3 lists the six styles specific to the `wx.ProgressDialog` that affect its behavior.

Table 9.3 Styles for `wx.ProgressDialog`

Style	Description
<code>wx.PD_APP_MODAL</code>	If this flag is set, the progress dialog is modal with respect to the entire application, meaning that it will block all user events. If the flag is not set, the progress dialog is modal only with respect to its parent window.
<code>wx.PD_AUTO_HIDE</code>	The progress dialog will automatically hide itself when it reaches its maximum value.
<code>wx.PD_CAN_ABORT</code>	Puts a Cancel button on the progress box for the user to stop the process. How to respond to a cancel from this dialog will be explained later.
<code>wx.PD_ELAPSED_TIME</code>	Displays the elapsed time that the dialog has been visible.

continued on next page

Table 9.3 Styles for `wx.ProgressDialog` (continued)

Style	Description
<code>wx.PD_ESTIMATED_TIME</code>	Displays an estimate of the total time to complete the process based on the amount of time already elapsed, the current value of the counter, and the maximum value of the counter.
<code>wx.PD_REMAINING_TIME</code>	Displays an estimate of the amount of time remaining in a process, or (estimated time – elapsed time).

To use the progress dialog, make a call to its only method, `Update(value, newmsg="")`. The `value` argument is the new internal value of the progress dialog, and calling `update` causes the progress bar to be redrawn based on the proportion between the new value and the maximum value set in the constructor. The `value` argument can be higher, lower, or equal to the current value argument. If the optional `newmsg` argument is included, the text message on the dialog changes to that string. This allows you to give the user a text description of the current progress.

The `Update()` method usually returns `True`. However, if the user has canceled the dialog via the Cancel button, the next time you `Update()`, the method will return `False`. This is your chance to respond to the user's request to cancel, presumably by stopping whatever process you are measuring. Given this mechanism for detecting a user cancel, it is recommended that you update the progress bar as often as possible, so you can test for the cancel.

9.2 Using standard dialogs

Most operating systems offer standard dialog boxes for tasks like file choosing, font selection, and color picking. This enables users to see a consistent look and feel across the entire platform. You can use these dialogs from wxPython to give your application the same advantage. In addition, if you use wxPython, it provides similar dialogs even on platforms that don't have system dialogs for the feature.

9.2.1 How can I use a file picker?

File-choosing dialogs tend to be consistent from application to application. In wxPython, the `wx.FileDialog` uses the native OS dialog for the major platforms, and uses non-native look-alikes for other operating systems. The MS Windows version is displayed in figure 9.6.

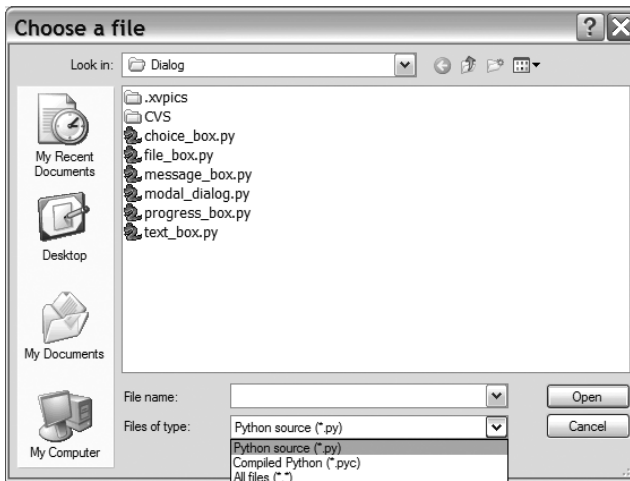


Figure 9.6
The standard Windows
file chooser

You can set up the file dialog to start in any directory, and you can also pass it a wildcard filter to limit the display to only certain file types. Listing 9.6 displays a basic example.

Listing 9.6 Using `wx.FileDialog`

```
import wx
import os

if __name__ == "__main__":
    app = wx.PySimpleApp()
    wildcard = "Python source (*.py)|*.py| \
              Compiled Python (*.pyc)|*.pyc| \
              All files (*.*)|*.*"
    dialog = wx.FileDialog(None, "Choose a file", os.getcwd(),
                          "", wildcard, wx.OPEN)
    if dialog.ShowModal() == wx.ID_OK:
        print dialog.GetPath()
    dialog.Destroy()
```

The file dialog is the most complex dialog we've seen in this chapter; in that it has several properties that can be programmatically read and written. The constructor allows you to set some of the properties, as in:

```
wx.FileDialog(parent, message="Choose a file", defaultDir="",
             defaultFile="", wildcard="*.*", style=0,
             pos=wx.DefaultPosition)
```

The message argument appears in the title bar of the window. The `defaultDir` tells the dialog what directory to display initially. If the argument is empty or

represents a directory that doesn't exist, the dialog starts in the current working directory. The `defaultFile` preselects a file, typically used when saving a file. The `wildcard` argument allows you to filter the list based on a given pattern, using the usual `*` and `?` as wildcard characters. The wildcard can either be a single pattern, such as `*.py`, or a set of patterns in the format `<description> | <pattern> | <description> | <pattern>`—similar to the wildcard used in listing 9.6.

```
"Python source (*.py)|*.py|Compiled Python (*.pyc)|*.pyc|
  All files (*.*)|*.*"
```

If there is a pattern with multiple entries, they display in the familiar pull-down menu shown in figure 9.6. The `pos` argument is not guaranteed to be supported by the underlying system.

Selecting a file

The two most important style flags for `wx.FileDialog` are `wx.OPEN` and `wx.SAVE`, which indicate the kind of dialog and affect the behavior of the dialog.

A dialog used for opening a file has two flags that further affect behavior. The flag `wx.HIDE_READONLY` causes the dialog to gray out the checkbox that allows the user to open the file in read-only mode. The flag `wx.MULTIPLE` allows the user to select multiple files in a single directory for opening.

Save file dialogs have one useful flag, `wx.OVERWRITE_PROMPT`, that forces the user to confirm saving a file if the file already exists.

Either kind of file dialog can use the `wx.CHANGE_DIR` flag. When this flag is raised, a file selection also changes the application's working directory to the directory where the selection took place. Among other things, this allows the next file dialog to open in the same directory without requiring that the application store that value elsewhere.

Unlike the other dialogs we've seen so far in this chapter, these properties are all gettable and settable via methods. This is true for the properties `directory`, `filename`, `style`, `message`, and `wildcard`, all of which have getters and setters using the usual `Get/Set` naming convention.

After the user has exited the dialog, and after checking that it was exited with `wx.OK`, you can get the user's choice by using the method `GetPath()`, which returns the full pathname of the file as a string. If the dialog is an open dialog with `wx.MULTIPLE` selected, use `GetPaths()` instead. That method returns a Python list of path strings. If for some reason you need to know which of the pull-down filters was active when the user made her selection, you can use the `GetFilterIndex()` method, which returns an integer index into the list. To change the index programmatically, use `SetFilterIndex()`.

The following is a convenience function for using file dialogs.

```
wx.FileSelector(message, default_path="", default_filename="",
               default_extension="", wildcard="*.**", flags=0, parent=None,
               x=-1, y=-1)
```

The `message`, `default_path`, `default_filename`, and `wildcard` arguments do what you'd expect from the constructor, despite being named differently. The `flags` argument is normally called `style`, and the `default_extension` adds an extension onto a selected file name that doesn't when you save a file. The return value is the string pathname if the user presses OK, or an empty string if the user presses Cancel.

Selecting a directory

If the user wants to select a directory rather than a file, use `wx.DirDialog`, which presents a tree view of the directory structure as shown in figure 9.7.

The directory selector is somewhat simpler than a file dialog. Listing 9.7 displays the relevant code.

Listing 9.7 Displaying a directory chooser dialog

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = wx.DirDialog(None, "Choose a directory:",
                        style=wx.DD_DEFAULT_STYLE | wx.DD_NEW_DIR_BUTTON)
    if dialog.ShowModal() == wx.ID_OK:
        print dialog.GetPath()
    dialog.Destroy()
```

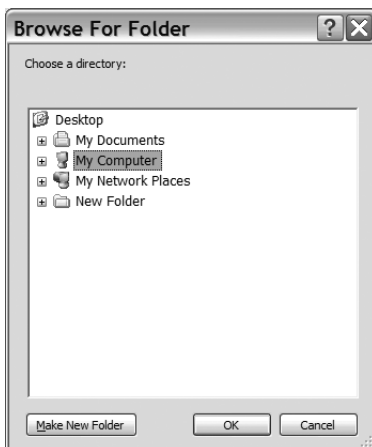


Figure 9.7
A directory selection dialog

Nearly all of the functionality of this dialog is in the constructor.

```
wx.DirDialog(parent, message="Choose a directory", defaultPath="",
             style=0, pos = wx.DefaultPosition, size = wx.DefaultSize,
             name="wxDirCtrl")
```

Because the message argument displays inside the dialog itself, you don't need a hook to change the title bar. The `defaultPath` tells the dialog what to select, and if it's empty, the dialog shows the root of the file system. The `pos` and `size` arguments are ignored under MS Windows, and the `name` argument is ignored in all operating systems. The style flag for this dialog, `wx.DD_NEW_DIR_BUTTON`, gives the dialog a button for creating a directory. This flag may not work in older versions of MS Windows.

The `path`, `message`, and `style` properties of this class have typical getters and setters. You can use the `GetPath()` method to retrieve the user selection after the dialog is dispatched. This dialog also has a convenience function.

```
wx.DirSelector(message=wx.DirSelectorPromptStr, default_path="",
              style=0, pos=wxDefaultPosition, parent=None)
```

All arguments are as in the constructor. The function returns the selected directory name as a string if OK is pressed, and the empty string if Cancel is pressed.

9.2.2 How can I use a font picker?

The font picker dialog in wxPython is different than the file dialog, because it uses a separate helper class to manage the information it presents. Figure 9.8 displays the MS Windows version of the font dialog.

Listing 9.8 displays the code used to generate figure 9.8, and should look somewhat different than previous dialog examples.

Listing 9.8 Sample code for a font picker dialog box

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = wx.FontDialog(None, wx.FontData())
    if dialog.ShowModal() == wx.ID_OK:
        data = dialog.GetFontData()
        font = data.GetChosenFont()
        colour = data.GetColour()
        print 'You selected: "%s", %d points\n' % (
            font.GetFaceName(), font.GetPointSize())
    dialog.Destroy()
```

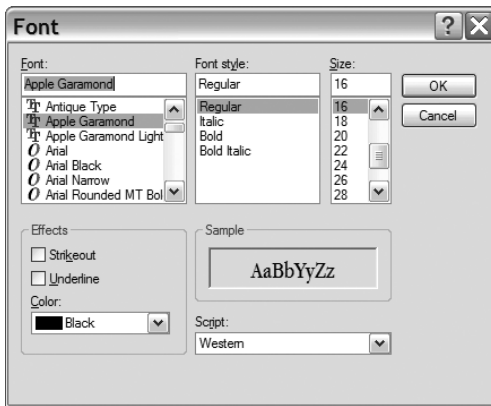


Figure 9.8
A sample font picker dialog

The constructor for `wx.FontDialog` is much simpler than the previous constructors.

```
wx.FontDialog(parent, data)
```

You cannot set a message or caption for this dialog, and the information that is ordinarily passed as style flags is instead contained in the `data` argument, which is of the class `wx.FontData`. The `wx.FontDialog` class has only one useful method of its own, which is `GetFontData()`, returning the font data instance.

The `wx.FontData` instance allows you to set the values that govern the display of the font dialog, and also contains the information entered by the user. For example, in listing 9.8 the code calls two getter methods of the `wx.FontData` instance to determine the details of the selected font. The constructor for `wx.FontData` takes no arguments—all properties must be set by using the methods in table 9.4

Table 9.4 Methods of `wx.FontData`

Method	Description
<code>GetAllowSymbols()</code> <code>SetAllowSymbols(allowSymbols)</code>	Determines whether symbol-only fonts (like dingbats) are displayed in the dialog. The argument is a Boolean. Only meaningful in Windows systems. The initial value of this property is <code>True</code> .
<code>GetChosenFont()</code> <code>SetChosenFont(font)</code>	Returns the font that the user has chosen as a <code>wx.Font</code> object. You should never need to call the setter for this property. If the user has selected Cancel, this property is <code>None</code> . The <code>wx.Font</code> class will be discussed in more detail in Chapter 12.

continued on next page

Table 9.4 Methods of `wx.FontData` (continued)

Method	Description
<code>GetColour()</code> <code>SetColour(colour)</code>	Returns the color selected in the color portion of the dialog. The setter allows you to preset the default value. The getter returns a <code>wx.Colour</code> instance. The setter can take one of those, or a string with the name of a color. The initial value of this property is <code>black</code> .
<code>GetEnableEffects()</code> <code>EnableEffects(enable)</code>	In the MS Windows version of the dialog, this property controls the appearance or nonappearance of controls to select color, strikethrough, and underline features of the font.
<code>GetInitialFont()</code> <code>SetInitialFont(font)</code>	Returns the font which is the initial value of the dialog (i.e., the current application font). This property should be explicitly set by the application before the dialog is displayed. Its initial value is <code>None</code> .
<code>SetRange(min, max)</code>	Sets the available range for the point size of the font. Only used on MS Windows systems. The initial values are 0 and 0, which means there are no limits on the range.
<code>GetShowHelp()</code> <code>SetShowHelp()</code>	If <code>True</code> , the MS Windows version of this dialog will display a Help button. The initial value is <code>False</code> .

A convenience function for the font dialog, which helpfully sidesteps the whole `wx.FontData` class, is as follows.

```
wx.GetFontFromUser(parent, fontInit)
```

The `fontInit` argument is an instance of `wx.Font` that is used as the initial value of the dialog. The return value of the function is a `wx.Font` instance. If the user closes the dialog with an OK, the method `wx.Font.Ok()` returns `True`, otherwise, it returns `False`.

9.2.3 How can I use a color picker?

The color picker dialog is similar to the font dialog, because it uses an external data class to manage its information. Figure 9.9 displays the MS Windows version of the dialog.

Listing 9.9 displays the code to generate the dialog, which is nearly identical to the code seen in the previous section for the font picker.

Listing 9.9 Code for a color picker dialog

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
```

```

dialog = wx.ColourDialog(None)
dialog.GetColourData().SetChooseFull(True)
if dialog.ShowModal() == wx.ID_OK:
    data = dialog.GetColourData()
    print 'You selected: %s\n' % str(data.GetColour().Get())
dialog.Destroy()

```

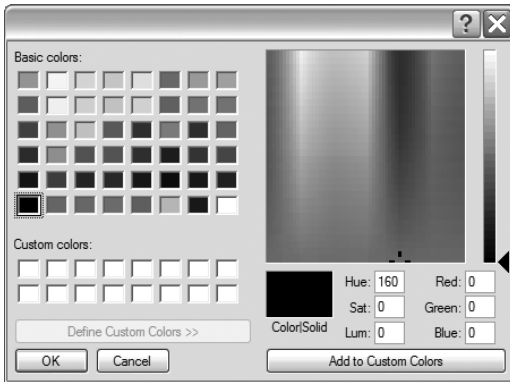


Figure 9.9
A standard wxPython color picker

The wxPython class for the color selector is `wx.ColourDialog`. Those of you in America will need to remember the non-USA spelling “colour.” For those of you outside America, I’m sure this is a welcome change of pace. The constructor is simple, without many options to tweak, as in the following:

```
wx.ColourDialog(parent, data=None)
```

The `data` argument is an instance of the class `wx.ColourData`, which is simpler than its font data counterpart. It contains only the default no-argument constructor, and the following three properties:

- `GetChooseFull/SetChooseFull(flag)` A Boolean property that works under MS Windows only. When set, the color picker shows the full dialog, including the custom color selector. When unset, the custom color selector is not shown.
- `GetColour/SetColour(colour)` The property is of the type `wx.Colour`. This is the color value selected by the user. After the graph is closed, call this getter to see the user selection. Initially it is set to black. If it is set before the dialog is displayed, the dialog initially displays this color.
- `GetCustomColour(i)/SetCustomColour(i, colour)` returns or sets the element in the custom color array with index `i`. The index is between 0 and 15. Initially all of the custom colors are white.

A simple convenience function bypasses the `wx.ColourData` entirely:

```
wx.GetColourFromUser(parent, colInit)
```

Where `colInit` is a `wx.Colour` instance and is the initial value of the dialog when displayed. The return value is also a `wx.Colour` instance. If the user closes the dialog with an OK, the method `wx.Colour.OK()` returns `True`. If the user closes it with a Cancel, the method returns `False`.

9.2.4 Can I allow the user to browse images?

If you are doing graphics manipulation in your program, it's often useful to provide the user with thumbnails of the images while they're browsing the file tree. A wxPython dialog for this purpose is called `wx.lib.imagebrowser.ImageDialog`. Figure 9.10 displays a sample.

Listing 9.10 displays the simple code for this image browser dialog.

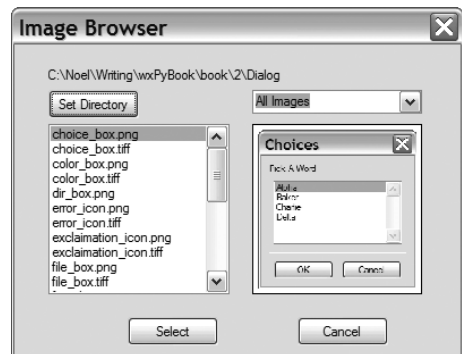


Figure 9.10 A typical image dialog browser

Listing 9.10 Creating an image browser dialog

```
import wx
import wx.lib.imagebrowser as imagebrowser

if __name__ == "__main__":
    app = wx.PySimpleApp()
    dialog = imagebrowser.ImageDialog(None)
    if dialog.ShowModal() == wx.ID_OK:
        print "You Selected File: " + dialog.GetFile()
    dialog.Destroy()
```

The `wx.lib.imagebrowser.ImageDialog` class is straightforward, and has relatively few options for the programmer to set. To change the dialog's behavior, review the Python source for changing the types of files displayed. The constructor takes just two arguments.

```
ImageDialog(parent, set_dir=None)
```

The `set_dir` argument is the directory in which the dialog when displayed. If it is not set, the application's current working directory is used. After the dialog is

closed, `GetFile()` returns the complete path string of the selected file, and `GetDirectory()` returns just the directory portion.

9.3 Creating a wizard

A *wizard* is a series of simple dialogs chained together to force the user to step through them one by one. Typically, they are used to guide a user through installation or a complex setup procedure by breaking down the information into small pieces. Figure 9.11 displays a sample wizard, displaying Back and Next buttons.

In wxPython, a wizard is a series of pages controlled by an instance of the class `wx.wizard.Wizard`. The wizard instance manages the events that take the user through the pages. The pages themselves are instances of either the class `wx.wizard.WizardPageSimple` or `wx.wizard.WizardPage`. In both cases, they are merely `wx.Panel` instances with the additional logic needed to manage the page chain. The difference between the two instances is manifested only when the user presses the Next button. An instance of `wx.wizard.WizardPage` allows you to determine dynamically which page to navigate to at runtime, whereas an instance of `wx.wizard.WizardPageSimple` requires that the order be preset before the wizard is displayed. Listing 9.11 displays the code for a simple wizard.



Figure 9.11 A simple wizard sample

Listing 9.11 Creating a simple static wizard

```
import wx
import wx.wizard

class TitledPage(wx.wizard.WizardPageSimple):
    def __init__(self, parent, title):
        wx.wizard.WizardPageSimple.__init__(self, parent)
        self.sizer = wx.BoxSizer(wx.VERTICAL)
        self.SetSizer(self.sizer)
        titleText = wx.StaticText(self, -1, title)
        titleText.SetFont(
            wx.Font(18, wx.SWISS, wx.NORMAL, wx.BOLD))
        self.sizer.Add(titleText, 0,
            wx.ALIGN_CENTRE | wx.ALL, 5)
        self.sizer.Add(wx.StaticLine(self, -1), 0,
            wx.EXPAND | wx.ALL, 5)
```

1 Creating sample pages

```

if __name__ == "__main__":
    app = wx.PySimpleApp()
    wizard = wx.wizard.Wizard(None, -1, "Simple Wizard")
    page1 = TitledPage(wizard, "Page 1")
    page2 = TitledPage(wizard, "Page 2")
    page3 = TitledPage(wizard, "Page 3")
    page4 = TitledPage(wizard, "Page 4")
    page1.sizer.Add(wx.StaticText(page1, -1,
                                "Testing the wizard"))
    page4.sizer.Add(wx.StaticText(page4, -1,
                                "This is the last page.))
    wx.wizard.WizardPageSimple_Chain(page1, page2)
    wx.wizard.WizardPageSimple_Chain(page2, page3)
    wx.wizard.WizardPageSimple_Chain(page3, page4)
    wizard.FitToPage(page1)
    if wizard.RunWizard(page1):
        print "Success"

```

Creating wizard instance

Creating wizard pages

2 Creating page chain

3 Sizing the wizard

4 Running the wizard

- ❶ For the purpose of populating a wizard, we create a simple little page that contains a static text title. Typically, you'd have some form elements here, and probably some data for the user to enter.
- ❷ The function `wx.wizard.WizardPageSimple_Chain()` is a convenience method that mutually calls `SetNext()` and `SetPrev()` of the two pages passed as arguments.
- ❸ Calling `FitToSize()` sizes the wizard based on the page passed as an argument, and also all the pages reachable from that page in the chain. Call this method only after the page chain has been created.
- ❹ The argument to this method is the page to start the wizard on. The wizard knows to close when it reaches a page that has no `Next` page. The `RunWizard()` method returns `True` if the user goes through the whole wizard and exits by pressing the `Finish` button.

Creating the `wx.wizard.Wizard` instance is the first part of using a wizard. The constructor looks similar to the following:

```

wx.wizard.Wizard(parent, id=-1, title=wx.EmptyString,
                bitmap=wx.NullBitmap, pos=wx.DefaultPosition)

```

In this example, the `parent`, `id`, `title`, and `pos` are as in `wx.Panel`. If set, the `bitmap` argument displays on each page. There is one style flag, `wx.wizard.WIZARD_EX_HELPBUTTON`, that causes a `Help` button to display. This is an extended flag, and must be set using the two-step creation process outlined in chapter 8.

Typically, you'll manage the size of the window by calling `FitToSize()` as displayed in line ❸ of listing 9.11, however, you can also set a minimal size by

calling `SetPageSize()` with a tuple or `wx.Size` instance. The `GetPageSize()` method returns the current size. In both cases, the size is only used for the part of the dialog reserved for individual pages, while the dialog as a whole will be somewhat larger.

You can manage the pages from within this class. The method `GetCurrentPage()` returns the page currently being displayed, and if the wizard is not currently being displayed, the method returns `None`. You can determine if the current page has a next or previous page by calling `HasNextPage()` or `HasPrevPage()`. Running the wizard is managed with the `RunWizard()` method, as described in line ④ of listing 9.11.

Wizards fire command events that you can capture for more specialized processing, as displayed in table 9.5. In all these cases, the event object itself is of the class `wx.wizard.WizardEvent`, which exposes two methods. `GetPage()` returns the `wx.WizardPage` instance which was active when the event was generated, rather than the instance that may be displayed as a result of the event. The method `GetDirection()` returns `True` if the event is a page change going forward, and `False` if it is a page change going backward.

Table 9.5 Events of `wx.wizard.WizardDialog`

Event	Description
<code>EVT_WIZARD_CANCEL</code>	Fired when the the user presses the Cancel button. This event may be vetoed using <code>Veto()</code> , in which case the dialog will not be dismissed.
<code>EVT_WIZARD_FINISHED</code>	Fired when the user presses the Finished button.
<code>EVT_WIZARD_HELP</code>	Fired when the user presses the Help button.
<code>EVT_WIZARD_PAGE_CHANGED</code>	Fired after the page has already been changed, to allow for postprocessing.
<code>EVT_WIZARD_PAGE_CHANGING</code>	Fired when the user has requested a page change, but it has not yet occurred. This event may be vetoed (if, for example, there is a required field that needs to be filled).

The `wx.wizard.WizardPageSimple` class is treated as though it were a panel. The constructor for the class allows you to set the Previous and Next pages, as in the following:

```
wx.wizard.WizardPageSimple(parent=None, prev=None, next=None)
```

If you don't want to set them in the constructor, you can use the `SetPrev()` and `SetNext()` methods. And if that's too much trouble, you can use `wx.wizard.`

`WizardPageSimple_Chain()`, which sets up the chaining relationship between two pages.

The complex version of wizard pages, `wx.wizard.WizardPage`, differs slightly. Rather than setting the `Previous` and `Next` explicitly, it defines handler methods that allow you to use more elaborate logic to define where to go next. The constructor is as follows:

```
wx.WizardPage(parent, bitmap=wx.NullBitmap, resource=None)
```

If set, the `bitmap` argument overrides the bitmap set in the parent wizard. The `resource` argument loads the page from a `wxPython` resource. To handle the page logic, override `GetPrev()` and `GetNext()` to return whatever you want the wizard to do next. A typical usage may be to dynamically determine the `Next` page based on user response to the current page.

9.4 Showing startup tips

Many applications use startup tips as a way of introducing users to program features they otherwise may not see. There is a very simple mechanism in `wxPython` for showing startup tips. Figure 9.12 displays a sample tip window.

Listing 9.12 displays the code.



Figure 9.12 A sample tip window with a helpful message.

Listing 9.12 Displaying a startup tip in five lines or less

```
import wx

if __name__ == "__main__":
    app = wx.PySimpleApp()
    provider = wx.CreateFileTipProvider("tips.txt", 0)
    wx.ShowTip(None, provider, True)
```

There are two convenience functions that govern the startup tips. The first creates a `wx.TipProvider`, as in the following:

```
wx.CreateFileTipProvider(filename, currentTip)
```

The `filename` attribute is the name of a file with the string tips. The `currentTip` is the index of the tip within the file to start with, and the first tip in the file is index 0. The application is responsible for storing that information between runs.

The tip file is a simple text file where each line is a new tip. Blank lines in the file are skipped, and lines beginning with # are considered comments, and are also skipped. Here is the tip file for this example.

```
You can do startup tips very easily.  
Feel the force, Luke.
```

The tip provider is an instance of the class `wx.PyTipProvider`. If you need more elaborate functionality, you can create your own subclass of `wx.TipProvider` and override the function `GetTip()`.

The function that displays the tip is `wx.ShowTip()`.

```
wx.ShowTip(parent, tipProvider, showAtStartup)
```

The `parent` is the parent window, if any, and the `tipProvider` is usually created from `wx.CreateFileTipProvider`. The `showAtStartup` argument controls whether the Show Tips At Startup checkbox is selected. It does not control whether the tips are actually displayed at startup, that's up to you. The return value of this function is the Boolean state of the Show Tips At Startup checkbox so that you can use that value the next time your application starts.

9.5 Using validators to manage data in a dialog

A *validator* is a special wxPython object that simplifies managing data in a dialog. When we discussed events in chapter 3, we mentioned briefly that if a widget has a validator, the validator can be automatically invoked by the event system. We've also seen `validator` as a parameter in the constructor of several of the wxPython widget classes, but we haven't yet discussed them.

The validator has three unrelated functions:

- Validates the data in the control before the dialog closes
- Automatically transfers data to and from the dialog
- Validates the data as the user types

9.5.1 How do I use a validator to ensure correct data?

A validator object is a subclass of `wx.Validator`. The parent class is abstract, and isn't used directly. Although there are a couple of predefined validator classes in the C++ wxWidget set, in wxPython, you will need to define your own validator classes. As we've seen in other cases, your Python classes need to inherit from a Python-specific subclass, `wx.PyValidator`, to be able to override all the parent

methods. A custom validator subclass must also override the method `Clone()`, which should return an identical copy of the validator.

A validator is attached to a specific widget in your system. That can be accomplished in one of two ways. First, if the widget allows it, the validator can be passed as an argument to the widget constructor. If the widget does not have a validator argument to its constructor, you can still attach a validator by creating a validator instance and calling the widget's `SetValidator(validator)` method.

To validate the data in the control, start by overriding the method `Validate(parent)` in your validator subclass. The `parent` argument is the parent window of the validator's widget, either the dialog or a panel. Use this to get the data from other widgets in the dialog if that's important, or you can ignore the argument altogether. You can use `self.GetWindow()` to get a reference to the widget being validated. The return value of your `Validate(parent)` method is a Boolean. A `True` value indicates to the rest of the system that the data in the validator's widget is valid. A `False` value indicates a problem. You are allowed to use `wx.MessageBox()` to display an alert from the `Validate()` method, but you shouldn't do anything else that could raise events in the wxPython application.

The return value of the `Validate()` method is important. It comes into play when you attempt to close a dialog using the OK button, (the button with an ID of `wx.ID_OK`). As part of the processing of the OK click, wxPython calls the `Validate()` function of any widget the dialog contains that has a validator. If any of those methods return `False`, the dialog will not close. Listing 9.13 displays a sample dialog with a validator that checks to see that all text controls have data.

Listing 9.13 A validator checking that all text controls have data

```
import wx

about_txt = """\
The validator used in this example will ensure that the text
controls are not empty when you press the Ok button, and
will not let you leave if any of the Validations fail."""

class NotEmptyValidator(wx.PyValidator):    ← Creating the validator subclass
    def __init__(self):
        wx.PyValidator.__init__(self)

    def Clone(self):
        """
        Note that every validator must implement the Clone() method.
        """
        return NotEmptyValidator()
```

```

def Validate(self, win): ❶ Using the validator method
    textCtrl = self.GetWindow()
    text = textCtrl.GetValue()

    if len(text) == 0:
        wx.MessageBox("This field must contain some text!", "Error")
        textCtrl.SetBackgroundColour("pink")
        textCtrl.SetFocus()
        textCtrl.Refresh()
        return False
    else:
        textCtrl.SetBackgroundColour(
            wx.SystemSettings_GetColour(wx.SYS_COLOUR_WINDOW))
        textCtrl.Refresh()
        return True

def TransferToWindow(self):
    return True

def TransferFromWindow(self):
    return True

class MyDialog(wx.Dialog):
    def __init__(self):
        wx.Dialog.__init__(self, None, -1, "Validators: validating")

        # Create the text controls
        about = wx.StaticText(self, -1, about_txt)
        name_l = wx.StaticText(self, -1, "Name:")
        email_l = wx.StaticText(self, -1, "Email:")
        phone_l = wx.StaticText(self, -1, "Phone:")

        name_t = wx.TextCtrl(self, validator=NotEmptyValidator())
        email_t = wx.TextCtrl(self, validator=NotEmptyValidator())
        phone_t = wx.TextCtrl(self, validator=NotEmptyValidator())

        # Use standard button IDs
        okay = wx.Button(self, wx.ID_OK)
        okay.SetDefault()
        cancel = wx.Button(self, wx.ID_CANCEL)

        # Layout with sizers
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(about, 0, wx.ALL, 5)
        sizer.Add(wx.StaticLine(self), 0, wx.EXPAND|wx.ALL, 5)

        fgs = wx.FlexGridSizer(3, 2, 5, 5)
        fgs.Add(name_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(name_t, 0, wx.EXPAND)
        fgs.Add(email_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(email_t, 0, wx.EXPAND)

```

Using the validator ❷

```

fgs.Add(phone_l, 0, wx.ALIGN_RIGHT)
fgs.Add(phone_t, 0, wx.EXPAND)
fgs.AddGrowbleCol(1)
sizer.Add(fgs, 0, wx.EXPAND|wx.ALL, 5)

btns = wx.StdDialogButtonSizer()
btns.AddButton(okay)
btns.AddButton(cancel)
btns.Realize()
sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

self.SetSizer(sizer)
sizer.Fit(self)

app = wx.PySimpleApp()

dlg = MyDialog()
dlg.ShowModal()
dlg.Destroy()

app.MainLoop()

```

- ❶ This method tests that the underlying control has some data. If it does not, the background color of the widget is changed to pink.
- ❷ In these lines, a new validator is attached to each text field in the dialog.

Figure 9.13 displays the dialog after attempting to close it with a blank field.

The code that explicitly tells the dialog to check the validators is not in the listing—it is a part of the wxPython event system. Another difference between dialogs and frames is that dialogs have the validator behavior built-in and frames do not. If you would like to use validators for validating controls not located in a dialog, call the parent window's `Validate()` method. If the `wx.WS_EX_VALIDATE_RECURSIVELY` extra style is set for the window, `Validate()` of all the child windows is also called. If any of the validations fail, `Validate` returns `False`. Next, we'll discuss how to use validators to transfer data.

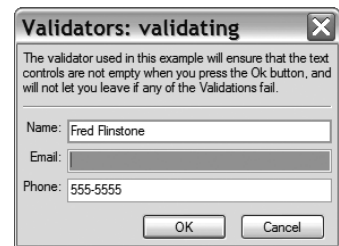


Figure 9.13 Attempting to close an invalid validator

9.5.2 How do I use a validator to transfer data?

The second important function of a validator is that it automatically transfers data into the dialog display when a dialog is opened, and automatically transfers data from the dialog to an external source when the dialog is closed. Figure 9.14 displays a sample dialog.

To accomplish this, you must override two methods in your validator subclass. The method `TransferToWindow()` is automatically called when the dialog is opened. You must use this method to put data into the validator's widget. The method `TransferFromWindow()` is automatically called when the dialog is closed using the OK button, after it has already been validated. You must use this method to move the data from the widget to some other source.

The fact that a data transfer must happen implies that the validator must know something about an external data object, as displayed in listing 9.14. In this example, each validator is initialized with a reference to a global data dictionary, and a key within that dictionary that is important to that control. When the dialog is opened, the `TransferToWindow()` method reads from the dictionary at that key and places the data in the text field. When the dialog is closed, `TransferFromWindow()` reverses the process and writes to the dictionary. The example displays a dialog box to show you the transferred data.

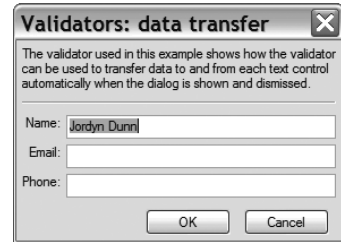


Figure 9.14 The transferring validator—this dialog will automatically display entered values when closed

Listing 9.14 A data transferring validator

```
import wx
import pprint

about_txt = """\
The validator used in this example shows how the validator
can be used to transfer data to and from each text control
automatically when the dialog is shown and dismissed."""

class DataXferValidator(wx.Validator):    ← Declaring the validator
    def __init__(self, data, key):
        wx.Validator.__init__(self)
        self.data = data
        self.key = key

    def Clone(self):
        """
```

```

        Note that every validator must implement the Clone() method.
        """
        return DataXferValidator(self.data, self.key)

def Validate(self, win):  ← Not validating data
    return True

def TransferToWindow(self):  ← Called on dialog open
    textCtrl = self.GetWindow()
    textCtrl.SetValue(self.data.get(self.key, ""))
    return True

def TransferFromWindow(self):  ← Called on dialog close
    textCtrl = self.GetWindow()
    self.data[self.key] = textCtrl.GetValue()
    return True

class MyDialog(wx.Dialog):
    def __init__(self, data):
        wx.Dialog.__init__(self, None, -1, "Validators: data transfer")

        about = wx.StaticText(self, -1, about_txt)
        name_l = wx.StaticText(self, -1, "Name:")
        email_l = wx.StaticText(self, -1, "Email:")
        phone_l = wx.StaticText(self, -1, "Phone:")

        name_t = wx.TextCtrl(self,  ← Associating a validator with widget
                             validator=DataXferValidator(data, "name"))
        email_t = wx.TextCtrl(self,
                             validator=DataXferValidator(data, "email"))
        phone_t = wx.TextCtrl(self,
                              validator=DataXferValidator(data, "phone"))

        okay = wx.Button(self, wx.ID_OK)
        okay.SetDefault()
        cancel = wx.Button(self, wx.ID_CANCEL)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(about, 0, wx.ALL, 5)
        sizer.Add(wx.StaticLine(self), 0, wx.EXPAND|wx.ALL, 5)

        fgs = wx.FlexGridSizer(3, 2, 5, 5)
        fgs.Add(name_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(name_t, 0, wx.EXPAND)
        fgs.Add(email_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(email_t, 0, wx.EXPAND)
        fgs.Add(phone_l, 0, wx.ALIGN_RIGHT)
        fgs.Add(phone_t, 0, wx.EXPAND)
        fgs.AddGrowableCol(1)
        sizer.Add(fgs, 0, wx.EXPAND|wx.ALL, 5)

```

```

        btns = wx.StdDialogButtonSizer()
        btns.AddButton(okay)
        btns.AddButton(cancel)
        btns.Realize()
        sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

        self.SetSizer(sizer)
        sizer.Fit(self)

app = wx.PySimpleApp()

data = { "name" : "Jordyn Dunn" }
dlg = MyDialog(data)
dlg.ShowModal()
dlg.Destroy()

wx.MessageBox("You entered these values:\n\n" +
              pprint.pformat(data))

app.MainLoop()

```

Calling of the transfer data methods of validators happens automatically for dialogs. To use validators for transferring data in non-dialog windows, call the parent widget's `TransDataFromWindow()` and `TransferDataToWindow()` methods as necessary. If the window has the `wx.WS_EX_VALIDATE_RECURSIVELY` extra style set, the transfer functions are also called on all of the child widgets.

In the next section, we'll discuss the most active use of a validator object, using it to evaluate data as the user enters it into the dialog box. This uses the validator and help from the wxPython event system.

9.5.3 How do I validate data as it is entered?

You can also use a validator to validate data entered into the dialog as the user enters it, before the data is passed to the widget. This is very powerful, since it can prevent bad data from getting into your application. Figure 9.15 displays an example, the dialog text explains the idea.

This method of validating data is less automated than other mechanisms. You must explicitly bind the character events from the validator's widget to a function, as in the following:

```
self.Bind(wx.EVT_CHAR, self.OnChar)
```

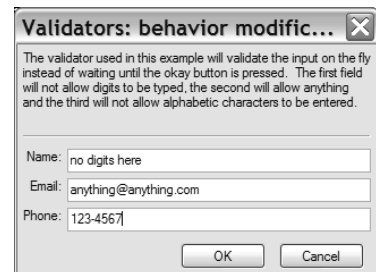


Figure 9.15 A validator verifying data on the fly

The widget assumes that the event source belongs to the validator. Listing 9.15 displays this binding in action.

Listing 9.15 Validating on the fly

```
import wx
import string

about_txt = """\
The validator used in this example will validate the input on the fly
instead of waiting until the okay button is pressed. The first field
will not allow digits to be typed, the second will allow anything
and the third will not allow alphabetic characters to be entered.
"""

class CharValidator(wx.PyValidator):
    def __init__(self, flag):
        wx.PyValidator.__init__(self)
        self.flag = flag
        self.Bind(wx.EVT_CHAR, self.OnChar)  ← Binding the character event

    def Clone(self):
        """
        Note that every validator must implement the Clone() method.
        """
        return CharValidator(self.flag)

    def Validate(self, win):
        return True

    def TransferToWindow(self):
        return True

    def TransferFromWindow(self):
        return True

    def OnChar(self, evt):
        key = chr(evt.GetKeyCode())
        if self.flag == "no-alpha" and key in string.letters:
            return
        if self.flag == "no-digit" and key in string.digits:
            return
        evt.Skip()

class MyDialog(wx.Dialog):
    def __init__(self):
        wx.Dialog.__init__(self, None, -1, "Validators: behavior
modification")

        # Create the text controls
```

**Viewing
the data
handler**

```

about = wx.StaticText(self, -1, about_txt)
name_l = wx.StaticText(self, -1, "Name:")
email_l = wx.StaticText(self, -1, "Email:")
phone_l = wx.StaticText(self, -1, "Phone:")

name_t = wx.TextCtrl(self, validator=CharValidator("no-digit"))
email_t = wx.TextCtrl(self, validator=CharValidator("any"))
phone_t = wx.TextCtrl(self, validator=CharValidator("no-alpha"))
okay = wx.Button(self, wx.ID_OK)
okay.SetDefault()
cancel = wx.Button(self, wx.ID_CANCEL)
sizer = wx.BoxSizer(wx.VERTICAL)
sizer.Add(about, 0, wx.ALL, 5)
sizer.Add(wx.StaticLine(self), 0, wx.EXPAND|wx.ALL, 5)

fgs = wx.FlexGridSizer(3, 2, 5, 5)
fgs.Add(name_l, 0, wx.ALIGN_RIGHT)
fgs.Add(name_t, 0, wx.EXPAND)
fgs.Add(email_l, 0, wx.ALIGN_RIGHT)
fgs.Add(email_t, 0, wx.EXPAND)
fgs.Add(phone_l, 0, wx.ALIGN_RIGHT)
fgs.Add(phone_t, 0, wx.EXPAND)
fgs.AddGrowableCol(1)
sizer.Add(fgs, 0, wx.EXPAND|wx.ALL, 5)

btns = wx.StdDialogButtonSizer()
btns.AddButton(okay)
btns.AddButton(cancel)
btns.Realize()
sizer.Add(btns, 0, wx.EXPAND|wx.ALL, 5)

self.SetSizer(sizer)
sizer.Fit(self)

app = wx.PySimpleApp()

dlg = MyDialog()
dlg.ShowModal()
dlg.Destroy()

app.MainLoop()

```

**Binding the
validator**

Because the `OnChar()` method is in a validator, it gets called before the widget responds to the character event. The method allows the event to pass on to the widget by using `skip()`. You must call `skip()`, otherwise the validator interferes with normal event processing. The validator performs a test to see if the character is valid for the control. If the character is invalid, `skip()` is not called, and event

processing stops. If necessary, events other than `wx.EVT_CHAR` can also be bound and the validator handles those events before the widget does.

Validators are a powerful and flexible mechanism for handling data in your wxPython application. Using them properly helps make the development and maintenance of your application smoother.

9.6 Summary

- Dialogs are used to handle interaction with the user in cases where there is a specific set of information to be obtained, and the interaction is usually over quickly. In wxPython, you can use the generic `wx.Dialog` class to create your own dialogs, or you can use one of several predefined dialogs. In many cases, commonly used dialogs also have convenience functions that make the use of the dialog easier.
- Dialogs can be displayed modally, meaning that all other user input within the application is blocked while the dialog is visible. A modal dialog is invoked by using the `ShowModal()` method, which returns a value based on whether the user pressed OK or Cancel to the dialog. Closing a modal dialog does not destroy it, and the same dialog instance can be used again.
- There are three generic simple dialogs available in wxPython. `wx.MessageDialog` displays an alert box or a yes/no question. `wx.TextEntryDialog` allows the user to enter text, and `wx.SingleChoiceDialog` gives the user a choice based on a list of items.
- When performing a long background task, you can use `wx.ProgressDialog` to display progress information to the user. The user can pick a file using the standard file dialog by using the `wx.FileDialog` class. There is a standard tree view which allows the user to pick a directory that is created using the `wx.DirDialog` class.
- You can access the standard font picker using `wx.FontDialog` and the standard color picker using `wx.ColorDialog`. In both cases, the dialog behavior and user response are controlled by a separate data class.
- To browse thumbnail images, use the wxPython-specific class `wx.lib.imagebrowser.ImageDialog`. This class allows the user to walk through her file system and select an image.
- You can create a wizard by using the `wx.wizard.Wizard` class to tie together a group of related dialog forms. The dialog forms are instances of either the class `wx.wizard.WizardSimplePage` or `wx.wizard.WizardPage`. The difference

is that the page to page path for a simple page needs to be laid out before the wizard is displayed, while the standard page allows you to manage that logic at runtime.

- Startup tips can easily be displayed using the functions `wx.CreateFileTipProvider` and `wx.ShowTip`.
- Validators are powerful objects that can automatically prevent a dialog from closing if the data entered is incorrect. They can also automatically transfer data between a dialog display and an external object, and can verify data entry on the fly.

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

54995

ISBN 1-932394-62-1