wxPython is a GUI toolkit for the Python programming language built on top of the cross-platform wxWidgets GUI libraries.

This book will provide you with the skills to build highly functional and native-looking user interfaces for Python applications in multiple operating system environments. By working through the recipes, you will gain insights into and exposure to creating applications using wxPython. There are recipes for beginners to get started in GUI programming as well as tips to help experienced users get more out of their applications. The recipes will take you from the most basic application constructs all the way through to the deployment of complete applications.

What this book will do for you...

- Create full featured user interfaces
- Design and develop custom controls
- Deploy and distribute wxPython applications to Windows, Macintosh OS X, Linux, and other UNIX-like environments
- Handle and respond to application events
- Manage and display data using grids
- Interact with web services from your GUI
- Use paint events to draw custom displays
- Support the display of user interfaces in multiple languages

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

wxPython is a GUI toolkit for the Python programming language built on top of the cross-platform wxWidgets GUI libraries.

This book will provide you with the skills to build highly functional and native-looking user interfaces for Python applications in multiple operating system environments. By working through the recipes, you will gain insights into and exposure to creating applications using wxPython. There are recipes for beginners to get started in GUI programming as well as tips to help experienced users get more out of their applications. The recipes will take you from the most basic application constructs all the way through to the deployment of complete applications.

What this book will do for you...

- Create full featured user interfaces
- Design and develop custom controls
- Deploy and distribute wxPython applications to Windows, Macintosh OS X, Linux, and other UNIX-like environments
- Handle and respond to application events
- Manage and display data using grids
- Interact with web services from your GUI
- Use paint events to draw custom displays
- Support the display of user interfaces in multiple languages

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

wxPython is a GUI toolkit for the Python programming language built on top of the cross-platform wxWidgets GUI libraries.

This book will provide you with the skills to build highly functional and native-looking user interfaces for Python applications in multiple operating system environments. By working through the recipes, you will gain insights into and exposure to creating applications using wxPython. There are recipes for beginners to get started in GUI programming as well as tips to help experienced users get more out of their applications. The recipes will take you from the most basic application constructs all the way through to the deployment of complete applications.

What this book will do for you...

- Create full featured user interfaces
- Design and develop custom controls
- Deploy and distribute wxPython applications to Windows, Macintosh OS X, Linux, and other UNIX-like environments
- Handle and respond to application events
- Manage and display data using grids
- Interact with web services from your GUI
- Use paint events to draw custom displays
- Support the display of user interfaces in multiple languages

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Over 80 step-by-step recipes to get you up to speed with building your own wxPython applications

Cody Precord

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'wxPython Starting Points'
- A synopsis of the book’s content
- More information on *wxPython Application Development Cookbook*
Cody Precord is a software engineer based in Minneapolis, MN, USA. He designs and writes systems and application software for Windows, AIX, Linux, and Macintosh OS X using primarily C++, C#, C, Perl, Bash, and Python. The constant need for working on multiple platforms naturally led Cody to the wxPython toolkit, which he has used for several years. He is the author of *wxPython 2.8 Application Development Cookbook*, Packt Publishing and has also contributed to the development of the wxPython library. Cody is interested in promoting cross platform development practices and improving usability in software.
In today's world of desktop applications, there is a great amount of incentive in being able to develop applications that can run in more than one environment. Currently, there are a handful of options available for cross platform frameworks to develop desktop applications in Python; wxPython is one such cross platform GUI toolkit for the Python programming language. It allows Python programmers to simply and easily create programs with a complete, highly functional graphical user interface. The wxPython code style has changed quite a bit over the years and has become much more Pythonic. The examples that you will find in this book are fully up to date and reflect this change in style. This cookbook provides you with the latest recipes to quickly create robust, reliable, and reusable wxPython applications. These recipes will guide you right from writing simple, basic wxPython scripts all the way through complex concepts and also feature various design approaches and techniques in wxPython.

This book starts off by covering a variety of topics, from the most basic requirements of a wxPython application to some of the more in-depth details of the inner workings of the framework, thus laying a foundation for any wxPython application. It then explains event handling, basic and advanced user interface controls, interface design and layout, creating dialogs, components, extending functionality, and so on. We will conclude by learning how to build and manage applications for distribution.

For each of the recipes, there is first an introductory and then more advanced examples along with plenty of example code that shows you how to develop and manage user-friendly applications. For more experienced developers, most recipes also include an additional discussion of the solution, allowing you to further customize and enhance the component.

What this book covers

Chapter 1, wxPython Starting Points, teaches the basics of getting started with building applications with wxPython.

Chapter 2, Common User Controls, introduces you to the commonly used UI components and how use them in wxPython.
Chapter 3, *UI Layout and Organization*, shows you how to lay out and present user controls on screen using Sizers.

Chapter 4, *Containers and Advanced Controls*, introduces you to various container-type and specialized controls, such as web browsers.

Chapter 5, *Data Displays and Grids*, shows you how to display and work with data using the Grids, Lists, and DataView controls.

Chapter 6, *Ways to Notify and Alert*, teaches you the techniques of alerting and notifying users with information.

Chapter 7, *Requesting and Retrieving Information*, shows you how to prompt users for information and input.

Chapter 8, *User Interface Primitives*, shows you how to use DeviceContexts to draw and customize your own UI components.

Chapter 9, *Creating and Customizing Components*, teaches you the techniques of designing and creating your own custom controls.

Chapter 10, *Getting Your Application Ready for Release*, shows you how to manage application configuration and build packages for release.
1

wxPython Starting Points

In this chapter, we will cover:

- Creating an application object
- Adding the main frame
- Using bitmaps
- Binding to events
- Understanding the hierarchy of the UI
- Controlling the propagation of events
- Accessing the clipboard
- Supporting drag and drop
- Handling AppleEvents

Introduction

In this chapter, we will take a quick overview on getting started with wxPython, including how to get an app started as well as handling events and supporting basic integration with various operating system features for the environments that the application may be operated in. These concepts are used throughout the recipes in this book as well as in any wxPython application you may develop. The recipes throughout this book target wxPython 3.0 running on Python 2.7. Many features exist and work in earlier versions of wxPython as well, but your mileage may vary with the recipes in this book when using a version earlier than 3.0.
**Creating an application object**

The App object is an object that all wxPython applications must create before any other GUI object. This object creates the application and provides its main event loop, which is used to dispatch events and connect actions in the UI with the actions in your programs.

This recipe will introduce how to create a minimal wxPython application, which will be used as foundation for every other recipe in this book.

**How to do it...**

Perform the following steps:

1. Make the script as follows:

   ```python
   import wx
   
   class MyApp(wx.App):
       def OnInit(self):
           wx.MessageBox("Hello wxPython", "wxApp")
           return True
   
   if __name__ == "__main__":
       app = MyApp(False)
       app.MainLoop()
   ```

2. Run the script and take a look at the result:

![wxApp](image)

**How it works...**

There are three things to take note of in this simple application: the first, we created a subclass of the wx.App object; the second, we overrode the OnInit method; and the third, we called the MainLoop method of the application object. These simple steps set up the base for any application.
The **OnInit** method is called by the application's **MainLoop** method when it is started and provides an entry point to start up the main logic and user interface of your application. In this example, we just used it to show a simple pop-up dialog box. The application's **MainLoop** method continues to run until the last window associated with the application is closed. The **OnInit** method must return **true** in order to continue the initialization of the **MainLoop** applications.

The **MainLoop** method processes and dispatches all the messages that are needed to present the UI and direct messages for user actions initiated with button clicks. When the **OK** button is clicked on the dialog, it sends a message that is dispatched by the **MainLoop** method to close the dialog. In this example, once the dialog has returned, **OnInit** will also return, and there will be no window objects remaining. So, the application's **MainLoop** method will return as well, and this script will exit.

**There's more...**

Though generally the **wx.App** object is created as we did in this example, the class constructor also has four optional keyword arguments that can be used to modify some of its behavior:

```python
wx.App(redirect=False, filename=None, useBestVisual=False, clearSigInt=True)
```

The four optional keyword arguments are as follows:

- **redirect**: If set to **True**, stdout is redirected to a debug window
- **filename**: If redirect is **True** and this is not **None**, then stdout can be redirected to a file specified by this argument
- **useBestVisual**: This specifies whether the application should try to use the best visuals provided by the underlying toolkit. (This has no effect on most systems.)
- **clearSigInt**: Setting this to **True** will allow the application to be terminated by pressing **Ctrl+C** from the command line.

**See also**

- The **Handling errors gracefully** recipe in Chapter 10, *Getting Your Application Ready for Release*, provides additional information on methods that can be overridden in **wx.App**.
Adding the main frame

Most applications have some sort of main window that they want to show to allow their users to interact with the software. In wxPython, this window is called a frame. The frame is the main top-level container and the base for building most user interfaces in wxPython. This recipe will show how to create a frame and add it to an application.

How to do it...

You can do this by performing the following steps:

1. Start by making a subclass of `wx.Frame` with the following code:

   ```python
class MyFrame(wx.Frame):
   def __init__(self, parent, title=""):  
   super(MyFrame, self).__init__(parent, title=title)

   # Set an application icon
   self.SetIcon(wx.Icon("appIcon.png"))

   # Set the panel
   self.panel = wx.Panel(self)
   ```

2. Next, create an instance of the frame and show it using the following code:

   ```python
class MyApp(wx.App):
   def OnInit(self):
   self.frame = MyFrame(None, title="Main Frame")
   self.frame.Show()
   return True
   ```

3. Run the script and take a look at what we made:
How it works...

The Frame class creates a top-level window that can be used to present any number of other controls. A frame can be created without any parent window and will remain in the application until it is dismissed by the user.

In this recipe, we set up a couple of items on the MyFrame class:

1. We called SetIcon to set the custom application icon on the title bar of the frame. This icon was created from the appIcon.png file, which exists in the same directory as the script.
2. We created a Panel object and set the frame as its parent object. A panel is a plain rectangular control used to contain other controls and is shown as the rectangular area inside the frame's borders. A panel must have a parent window in order to be created.

Finally, in the App object's OnInit method, we created an instance of the frame specifying the title that we wanted to show in the frame's title bar and then called its Show method to display it on the screen. This recipe can be used as the preparation to create any wxPython application.

There's more...

The wx.Frame constructor has several style flags that can be specified in its constructor to modify its behavior and appearance.

<table>
<thead>
<tr>
<th>Style flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wx.DEFAULT_FRAME_STYLE</td>
<td>This flag is a bit mask of all the other flags described in the following sections</td>
</tr>
<tr>
<td>wx.MINIMIZE_BOX</td>
<td>This displays the minimize button on the title bar</td>
</tr>
<tr>
<td>wx.MAXIMIZE_BOX</td>
<td>This displays the maximize button on the title bar</td>
</tr>
<tr>
<td>wx.RESIZE_BORDER</td>
<td>This allows the frame to be resized by the user</td>
</tr>
<tr>
<td>wx.CAPTION</td>
<td>This displays a title caption on the frames title bar</td>
</tr>
<tr>
<td>wx.CLOSE_BOX</td>
<td>This displays the close button on the title bar</td>
</tr>
<tr>
<td>wx.SYSTEM_MENU</td>
<td>This displays a system menu (the menu that appears when clicking on the frame icon on Windows)</td>
</tr>
<tr>
<td>wx.CLIP_CHILDREN</td>
<td>This eliminates the flicker caused by background repaint (Windows only)</td>
</tr>
</tbody>
</table>

These style flags can be passed in any combination using a bitwise operator to turn off any of the features that you may not want to provide on all frames. Multiple flags can be combined using a bitwise or operation.
Using bitmaps

Bitmaps are the basic data type used to represent images in an application. The `wx.Bitmap` object can seamlessly load and decompress most common image file formats into a common representation that is usable by many UI controls. Adding bitmaps to the controls can make a UI more intuitive and easier to use.

How to do it...

Perform the following steps:

1. Let's start this time by making a subclass of `wx.Panel` to use as the container for the control that will show our bitmap on the screen, as follows:
   ```python
class ImagePanel(wx.Panel):
    def __init__(self, parent):
        super(ImagePanel, self).__init__(parent)
        # Load the image data into a Bitmap
        theBitmap = wx.Bitmap("usingBitmaps.png")

        # Create a control that can display the bitmap on the screen.
        self.bitmap = wx.StaticBitmap(self, bitmap=theBitmap)
   ```

2. Next, we will create an instance of the panel in a frame:
   ```python
class MyFrame(wx.Frame):
    def __init__(self, parent, title=""):  
        super(MyFrame, self).__init__(parent, title=title)

        # Set the panel
        self.panel = ImagePanel(self)
   ```

3. Run it and see the image shown on the panel:
Chapter 1

How it works...

The Bitmap class is used to load image data from the usingBitmaps.png file, which is located in the same directory as the script. This loads the PNG file data into a bitmap object, which can be used by the controls.

In this example, we used the StaticBitmap control, which is one of the easiest ways to display a bitmap in the UI. This control takes the bitmap data object and handles drawing it on the screen.

There's more...

In this recipe, we used a PNG file as the source for the bitmap, but wx.Bitmap also supports a wide range of other image formats, such as BMP, GIF, ICO, ICON, JPEG, TIF, XBM, XPM, and several others, depending on the build of wxWidgets used by wxPython.

Binding to events

wxPython is an event-driven framework; this means that all actions and the running of the UI is driven by events. Events are fired by objects to indicate that something has happened or needs to happen. MainLoop then dispatches these events to callback methods that are registered to be notified of the event. This recipe will show how to bind callback functions to events.

How to do it...

Perform the following functions:

1. First, start by creating a frame and binding to some of its events with the following code:

```python
class MyApp(wx.App):
    def OnInit(self):
        self.frame = wx.Frame(None, title="Binding Events")

        # Bind to events we are interested in
        self.frame.Bind(wx.EVT_SHOW, self.OnFrameShow)
        self.frame.Bind(wx.EVT_CLOSE, self.OnFrameExit)

        # Show the frame
        self.frame.Show()
        return True
```

```python
# Define events handling functions

def OnFrameShow(event):
    print("Frame Show")

def OnFrameExit(event):
    print("Frame Exit")
```
2. Next, define the event handler callback methods we specified in the `Bind` calls. These will get executed when the bound event occurs, as follows:

```python
def OnFrameShow(self, event):
    theFrame = event.EventObject
    print("Frame (%s) Shown!" % theFrame.Title)
    event.Skip()

def OnFrameExit(self, event):
    theFrame = event.EventObject
    print("Frame (%s) is closing!" % theFrame.Title)
    event.Skip()
```

**How it works...**

In the `OnInit` method, we created a frame object and then called `Bind` on it two times in order to bind our own two callback methods to these events that the frame emits. In this case, we bound to `EVT_SHOW` and `EVT_CLOSE`; these two events will be emitted by a window when the window transitions from being hidden to shown on screen and then when it is closed. Binding to events allows us to add some application-specific response when these two events occur. Now, our app's `OnFrameShow` and `OnFrameExit` callbacks will be executed by the framework in response to the event and allow us to print our log messages.

The first event, `EVT_SHOW`, happens as part of when the `Show` method is called on the frame in the app's `OnInit` method. The other event, `EVT_CLOSE`, occurs when the frame's close button is clicked on.

The event handler methods used in `Bind` always take one argument, which is an event object. This object is passed into the handler by the framework when it is called. The event object contains information about the event, such as a reference to the object that emitted it and other state information depending on what type of event was emitted.

**There's more...**

The `Bind` function can also take some additional optional parameters to set more fine-grain control on when the callback should be executed, as follows:

```python
Bind(event, handler, source=None, id=-1, id2=-1)
```

The arguments to this function are described as follows:

- `event`: This is the event to bind to.
- `handler`: This is the event handler callback function to bind.
source: This can be used to specify the window object that is the source of the event. If specified, then only when the source object generates the event will the handler be executed. By default, any event of the type that gets to the control will cause the handler to execute.

id1: This is used to specify the source object's ID instead of using the instance.

id2: When specified with id1, this can be used to specify a range of IDs to bind to.

There are many kinds of events that can be bound to depending on the type of control. The wxPython and wxWidgets online documentation provides a fairly complete list of events that are available for each control in the library. Note that the documentation is based on object hierarchy, so you may have to look to the base classes of an object to find the more general events that many controls share. You can find the documentation at http://wxpython.org/onlinedocs.php.

See also

- Take a look at the Controlling the propagation of events recipe section in this chapter for information on the behavior of events.

Understanding the hierarchy of the UI

There are certain rules and requirements to create a user interface; in its most fundamental form, the UI is just a collection of rectangles contained within other rectangles. This recipe will discuss how the hierarchy of controls is linked together.

How to do it...

You need to perform the following steps:

1. Let's start by defining the top-level window that resides at the top of the hierarchy:
   ```python
class MyFrame(wx.Frame):
    def __init__(self, parent, title=''):  
        super(MyFrame, self).__init__(parent, title=title)

        self.panel = MyPanel(self)
```

2. Next, let's define the Panel class, which will serve as the general container for user controls and give it a child control through the following code:
   ```python
class MyPanel(wx.Panel):
    def __init__(self, parent):
        super(MyPanel, self).__init__(parent)

        self.button = wx.Button(self, label='Push Me')
```
How it works...

All controls have an argument for a parent in their constructor. The parent is the container that the child control belongs to; so, in the first snippet when the MyPanel object was created, it was passed into the Frame object as its parent. This caused the panel rectangle to be placed inside of the rectangle of the Frame object. Then again, inside of the MyPanel object, a Button object was created with Panel as the parent, which instructed the button rectangle to be positioned inside the area owned by Panel.

There are three layers of containment in the window hierarchy for different categories of control types:

- **Top-level Windows (Frames and Dialogs):** These cannot be contained by any type of container when displayed on screen. They are always at the top of the visual hierarchy.
- **General Containers (Panels, Notebooks, and so on):** These are general container windows that serve the purpose of grouping other controls together and providing layout. They can contain other general containers or controls.
- **Controls (Buttons, CheckBoxes, ComboBoxes, and so on):** These are user controls that cannot contain any other controls. They are the leaves at the bottom of the tree.

There's more...

When building an application with a user interface, this hierarchy is important to remember as it plays a critical role in how the layout and design of the interface is performed. Most notably, in the middle general containers layer, the nesting and composition of the control layout in combination with event handling can lead to unexpected issues if this hierarchy is forgotten. So, just remember this tree structure when building out your application's interface:
Controlling the propagation of events

There are two main types of events in wxPython:

- Normal events
- Command events

Understanding how these events travel through the framework is important to understanding how to develop an application in this event-driven framework. This recipe will develop an example to show how to control the way an event is propagated.

How to do it...

The following steps can help us:

1. Let's start by creating a panel that has two buttons in it, by creating the following class:

   ```python
   class MyPanel(wx.Panel):
       def __init__(self, parent):
           super(MyPanel, self).__init__(parent)

           sizer = wx.BoxSizer()
           self.button1 = wx.Button(self, label="Button 1")
           sizer.Add(self.button1)
           self.button2 = wx.Button(self, label="Button 2")
           sizer.Add(self.button2)
           self.SetSizer(sizer)
           self.Bind(wx.EVT_BUTTON, self.OnButton)
   ```

2. Next, let's define the event handler for the panel to handle the button events as follows:

   ```python
   def OnButton(self, event):
       button = event.EventObject
       print("Button (%s) event at Panel!") % button.Label
       if button is self.button1:
           event.Skip()
   ```
3. In the next layer, let’s make a frame to hold the panel and also set it up to catch button events through the following code:

```python
class MyFrame(wx.Frame):
    def __init__(self, parent, title=""):  
        super(MyFrame, self).__init__(parent, title=title)

        self.panel = MyPanel(self)

        self.Bind(wx.EVT_BUTTON, self.OnButton)

    def OnButton(self, event):
        button = event.EventObject
        print("Button (%s) event at Frame!" % button.Label)
        event.Skip()
```

4. Finally, let’s do the same thing at the app level:

```python
class MyApp(wx.App):
    def OnInit(self):
        self.frame = MyFrame(None, title="Event Propagation")
        self.frame.Show();

        self.Bind(wx.EVT_BUTTON, self.OnButton)
        return True

    def OnButton(self, event):
        button = event.EventObject
        print("Button (%s) event at App!" % button.Label)
        event.Skip()```

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

**How it works...**

When you run this application and click on each of the buttons, you should see two distinct differences in behavior between how the events propagate. When clicking on the first button, the event handlers from the panel to the frame and finally to the app will be executed, whereas when clicking the second button, only the event handler at the panel is executed.
The EVT_BUTTON object is a command event, meaning it will propagate upward until it is stopped or it reaches the app. In this example, the second button is only propagated to the Panel handler because we called event.Skip when the event originated from the first button. Calling the Skip method tells the framework to propagate the event to the next handler in the chain, whereas not calling the Skip method tells the framework that the event has been handled and does not require further processing. So, it's important to know when to call Skip and when not to as sometimes it is necessary for the event to propagate to the base default handler in order for additional processing to occur. Try going back to the Binding to events recipe and removing the call to Skip in OnFrameExit to ensure that it prevents the frame from closing.

There's more...

As discussed at the beginning of this recipe, there are two types of events. This recipe only explored the more common types of command events that were propagated. Normal events stay local to where they are generated and do not propagate.

You can also create your own custom events if you want to use the event loop to pass messages using the newevent module in wx.lib, as follows:

```python
import wx
import wx.lib.newevent

# Create a special event
MyEvent, EVT_MYEVENT = wx.lib.newevent.NewCommandEvent();
```

This creates a new event object type and event binder object. The EVT_MYEVENT binder object can be bound to as any other event. Then, the MyEvent event object can be emitted through the use of the wx.PostEvent function, which sends the instance of the event through the event handler chain.

Accessing the clipboard

The clipboard is a system resource used to pass user data between applications in the operating system. This is most often associated with copying and pasting actions in an application. This recipe will show some basics on putting and getting data from the clipboard.

How to do it...

1. Let's first define a helper function to get some text from the clipboard, as follows:

```python
def GetClipboardText():
    text_obj = wx.TextDataObject()
    rtext = ""
    if wx.TheClipboard.IsOpened() or wx.TheClipboard.Open():
```
wxPython Starting Points

if wx.TheClipboard.GetData(text_obj):
    rtext = text_obj.GetText()
wx.TheClipboard.Close()
return rtext

2. Now, let's do the reverse and define a helper function to put text into the clipboard, as done with the following code:

def SetClipboardText(text):
    data_o = wx.TextDataObject()
    data_o.SetText(text)
    if wx.TheClipboard.IsOpened() or wx.TheClipboard.Open():
        wx.TheClipboard.SetData(data_o)
wx.TheClipboard.Close()

How it works...

Both functions work by creating TextDataObject, which provides a platform-independent way to represent the systems' native data format. Then, TheClipboard object is opened, and it is used to either get data from the clipboard of the given type or put data in the clipboard from the application. This can be boiled down to a simple three step process for any clipboard interaction:

1. Open clipboard
2. Set or get DataObject
3. Close the clipboard

Closing the clipboard after using is very important; it may prevent other processes from accessing it. The clipboard should only be kept open momentarily.

There's more...

The clipboard supports many other datatypes besides plain text, which can be used based on the situation and needs of the application.

<table>
<thead>
<tr>
<th>Datatypes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wx.BitmapDataObject</td>
<td>This is a bitmap data from the clipboard (drag and drop)</td>
</tr>
<tr>
<td>wx.CustomDataObject</td>
<td>This is a base class to represent application-specific data</td>
</tr>
<tr>
<td>wx.DataObjectSimple</td>
<td>This is a base class to create other data object types</td>
</tr>
<tr>
<td>wx.DataObjectComposite</td>
<td>This is a base class to support multiple formats</td>
</tr>
<tr>
<td>wx.FileDataObject</td>
<td>This is the data object for filenames</td>
</tr>
<tr>
<td>wx.HTMLDataObject</td>
<td>This is the HTML-formatted text container</td>
</tr>
<tr>
<td>wx.URLDataObject</td>
<td>This is the URL container data object</td>
</tr>
</tbody>
</table>
See also

- Take a look at the next recipe in this chapter, Supporting drag and drop, for more on how clipboard data objects can be used to transfer data between controls.

Supporting drag and drop

Many applications allow users to open files by dragging a file from the operating system and dropping it in the application. wxPython, of course, provides support for this as well, through its controls using DropTargets. This recipe will show how to set up a DropTarget to allow handling the dragging and dropping of files in an application.

How to do it...

1. First, let’s create a drop target object to accept files that are dragged over and dropped in the application with the following code:

   ```python
   class MyFileDropTarget(wx.FileDropTarget):
       def __init__(self, target):
           super(MyFileDropTarget, self).__init__()
           self.target = target

       def OnDropFiles(self, x, y, filenames):
           for fname in filenames:
               self.target.AppendText(fname)
   ```

2. Next, all that is left is to connect the drop target to the window that should accept the dropped file(s). An example of this is shown in the following code:

   ```python
   class MyFrame(wx.Frame):
       def __init__(self, parent, title=""):  
           super(MyFrame, self).__init__(parent, title=title)

           # Set the panel
           self.text = wx.TextCtrl(self, style=wx.TE_MULTILINE)
           self.text.AppendText("Drag and drop some files here!")
           dropTarget = MyFileDropTarget(self.text)
           self.text.SetDropTarget(dropTarget)
   ```
How it works...

Drag and drop functions with the use of DropSources and DropTargets. In this case, we wanted to allow files to be dropped in the application, so FileDropTarget was created and associated with the TextCtrl window. DropTargets have several virtual callback functions that can be overridden to intercept different actions during the drag and drop events. As FileDropTarget is specialized for files, it only required overriding OnDropFiles, which is called with the list of filenames that were dropped in the application. It is necessary to subclass the drop target in order to intercept and handle the data it receives.

In order for a window to accept drag and drop actions, it must have a DropTarget set; the DropTarget then gives feedback on whether the data can be accepted or not as well as handling the reception of the data. Try out the example code with this recipe and you will see the mouse cursor change as you drag a file over; then, try again by dragging some text from another application to see the difference.

There's more...

It's also possible to create more application-specific drag and drop handling if needed for custom datatypes by deriving a custom drop target class from PyDropTarget. This class provides several more overridable methods to allow the handling of various events during the action. Here's a table explaining this:

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnEnter(x, y, dragResult)</td>
<td>This is called when a drag object enters the window. The dragResult value returned from this method sets the custom cursor to provide feedback to the user (wx.DragCancel, wx.DragCopy, and so on).</td>
</tr>
<tr>
<td>OnDragOver(x, y, dragResult)</td>
<td>This is called while the object is dragged over the window. It returns a dragResult to give visual feedback.</td>
</tr>
<tr>
<td>OnLeave()</td>
<td>This is called when the drag object leaves the window.</td>
</tr>
<tr>
<td>OnDrop(x, y)</td>
<td>This is called when the drag object is dropped in the window. This method should return True if the data is accepted.</td>
</tr>
<tr>
<td>OnData(x, y, dragResult)</td>
<td>This is called after the data is accepted in OnDrop. The dropped data object is contained in drop targets data object (refer to GetDataObject). This should then typically just return the default passed in dragResult.</td>
</tr>
</tbody>
</table>
Handling AppleEvents

AppleEvents are special kinds of high-level system events used by the OS X operating system to pass information between processes. In order to handle system events, such as when a file is dropped in the application's Dock icon, it's necessary to handle AppleEvents. Implementing the handlers for these methods can allow your app to behave more natively when run on OS X.

This recipe is specific to the OS X operating system and will have no effect on other operating systems.

How to do it...

Perform the following steps:

1. Define an app object in which we will override the available AppleEvent handlers through the following code:
   ```python
class MyApp(wx.App):
    def OnInit(self):
        self.frame = MyFrame("Apple Events")
        self.frame.Show()
        return True
```

2. Override the handlers that are available to deal with the opening of files:
   ```python
def MacNewFile(self):
    """Called in response to an open-application event""
    self.frame.AddNewFile()

def MacOpenFiles(self, fileNames):
    """Called in response to an openFiles message in Cocoa or an open-document event in Carbon""
    self.frame.AddFiles(fileNames)
```

3. Finally, let's also override the remaining available AppleEvent handlers that are defined in `wx.App` and have them redirected to some actions with our application's main window. The following code can help us do this:
   ```python
def MacOpenURL(self, url):
    """Called in response to get-url event""
    self.frame.AddURL(url)
```
wxPython Starting Points

    def MacPrintFile(self, fileName):
        """Called in response to a print-document event"""
        self.frame.PrintFile(fileName)

    def MacReopenApp(self):
        """Called in response to a reopen-application event"""
        if self.frame.IsIconized():
            self.frame.Iconize(False)
        self.frame.Raise()

How it works...

In the Carbon and Cocoa builds of wxPython, these additional Mac-specific virtual overrides are available for apps to implement. The app object has special handling for these events and turns them into simple function calls that can be overridden in derived applications to provide an app-specific handling of them.

The first two methods that we overrode are called in response to creating a new file or opening existing files, such as when a file is dragged and dropped in the application icon in the dock. The MacOpenFiles method is new since wxPython 2.9.3 and should be used instead of the MacOpenFile method that was provided in previous versions.

The other method that most apps should implement in some form is the MacReopenApp method. This method is called when a user clicks on the application icon in the dock. In this implementation, we ensure that the app is brought back to the foreground in response to this action.

There's more...

If there are other OS X-specific actions you want your app to handle, it is also possible to add support for additional AppleEvents to a wxPython application. It is not a particularly easy task as it requires writing a native extension module to catch the event, block the event loop, and then restore the Python interpreter's state back to wx after handling the event. There is a pretty good example that can be used as a starting point on the wxPython Wiki page (refer to [http://wiki.wxpython.org/Catching%20AppleEvents%20in%20wxMAC](http://wiki.wxpython.org/Catching%20AppleEvents%20in%20wxMAC)) if you find yourself needing to venture down this route.
Where to buy this book

You can buy wxPython Application Development Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.