
Creating GUI Applications with wxPython
A Project-Based Approach
Michael Driscoll

Creating GUI Applications with wxPython

Michael Driscoll

This book is for sale at http://leanpub.com/creatingapplicationswithwxpython

This version was published on 2020-12-07

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Welcome to **Creating GUI Applications with wxPython**. In this book, we will learn how to create several different desktop applications using the wxPython GUI toolkit. Each of the applications that we create will run on Windows, Mac and Linux. All of the code is open source and free for you to use and change at will. We will start off the book with a quick introduction to the wxPython framework itself and then we will jump into learning how to create fun little applications.

At the end of the book, I will discuss how you can distribute your applications to users.

This book will be using **Python 3.7** and **wxPython 4**.

Who is this book for?

This book is for anyone who would like to learn how to create cross-platform graphical user interfaces with Python. You should already know the Python programming language and it would help if you already know something about event-driven programming and object oriented programming. This book isn't really an intro to wxPython either. Instead, we will spend most of the chapters creating simple and functional applications and improving them. This will help you understand how all the pieces fit together when you are creating your own applications or enhancing the ones in this book.

If you would like an introduction to wxPython, you can see the following resources:

- wxPython documentation - https://wxpython.org/Phoenix/docs/html/index.html
- zetcode's wxPython tutorial - http://zetcode.com/wxpython/

About the Author

Mike Driscoll has been programming with the Python language for more than a decade. He has also been an active user and documenter of the wxPython GUI toolkit for almost as long as he's been using Python. When Mike isn't programming for work, he writes about Python on his blog: https://www.blog.pythonlibrary.org/. He has worked with Packt Publishing and No Starch Press as a technical reviewer for their books. He has also written several books.

You can see a full listing here:

- https://www.blog.pythonlibrary.org/books/

Conventions

As with most technical books, this one includes a few conventions that you need to be aware of. New topics and terminology will be in **bold**.

Code examples will look like the following:

```
import wx

app = wx.App(False)
frame = wx.Frame(None, title='Test')
frame.Show()
app.MainLoop()
```

Most code examples should work if you were to copy and paste them into your code editor, unless we are looking at a smaller portion of code explicitly.

Requirements

You will need a working version of the Python programming language to use this book. This book's examples were written using Python 3.6 and 3.7. If you do not have Python 3, you can get it here:

- https://python.org/download/

The wxPython package supports Python 3.4 - 3.7. The wxPython toolkit does **not** come included with Python. However you can install it with pip:

```
pip install wxPython
```

Some people recommend installing 3rd party packages such as wxPython using the following syntax:

```
python3 -m pip install wxPython --user
```

This will cause whichever version of Python 3 is mapped to your **python3** shortcut to install wxPython to itself. When you just run pip by itself, it is not always clear where the package will be installed.

> Note: Linux users may need to install some dependencies before pip can install wxPython. See the README file on the wxPython Github page for additional details. There is also an Extras directory on the wxPython website that has pre-built wheels for certain flavors of Linux here: https://extras.wxpython.org/wxPython4/extras/linux/. If you go that route, then you can install wxPython by using the following command: `pip install -U -f URL/to/wheel`

If you prefer to install packages into a Python virtual environment, you can use Python 3's `venv` package or the 3rd party package to do so. For more information on Python virtual environments, see the following URL:

- https://docs.python-guide.org/dev/virtualenvs/

Another method for installing into a Python virtual environment is the use of **pipenv**. The pipenv package is basically pip+virtualenv as an all-in-one tool.

You can use it like this:

```
pipenv install wxPython
pipenv shell
```

Any additional requirements will be explained later on in the book.

> Note: If you are using **Anaconda**, you may see a message like this when attempting to run wxPython: *This program needs access to the screen. Please run with a Framework build of python, and only when you are logged in on the main display of your Mac.* This occurs because you need to use **pythonw** when running wxPython, so you may need to adjust your settings in Anaconda to make it work correctly.

Integrated Development Environments (IDEs)

Python comes with an Integrated Development Environment called IDLE. However it can be a bit buggy to use when working with wxPython or PyQt. I usually use WingIDE Pro as my Python IDE of choice. However there are a couple of other popular options:

- PyCharm
- VS Code

Some developers like to use Vim or Sublime Text as well. There are benefits to using a Python-specific IDE though:

- Better syntax highlighting
- Built-in linters
- Better auto-complete
- Debugging

Both PyCharm and WingIDE have community editions that are free to use. VS Code is also free to use and has plugins for most popular programming languages. All three of these programs work on all major platforms as well.

You can also check out the following link for additional options:

- https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

Now you should be all set up and ready to start programming.

Book Source Code

The book's source code can be found on Github:

- https://github.com/driscollis/applications_with_wxpython

Reader Feedback

I welcome feedback about my writings. If you'd like to let me know what you thought of the book, you can send comments to the following address:

- comments@pythonlibrary.org

Errata

I try my best not to publish errors in my writings, but it happens from time to time. If you happen to see an error in this book, feel free to let me know by emailing me at the following:

- errata@pythonlibrary.org

Now let's get started!

Chapter 1 - An Intro to wxPython

The wxPython toolkit is a cross-platform Graphical User Interface (GUI) Framework that is currently maintained primarily by Robin Dunn. It provides Python bindings to the underlying wxWidgets toolkit, which is written in C++. The wxPython toolkit was first released in 1998 and is very stable. The community surrounding wxPython is quite nice and very welcoming and helpful to newcomers. You will probably see the term **Phoenix** in reference to **wxPython 4**. Phoenix was the code name for the Python 3 port of wxPython. The previous versions of wxPython are not completely compatible with wxPython 4.

If you have legacy wxPython applications or you find an old example that you want to convert to wxPython 4, then you will want to consult the following two URLs:

- Classic vs Phoenix:
  - https://wxpython.org/Phoenix/docs/html/classic_vs_phoenix.html
- wxPython Project Phoenix Migration Guide:
  - https://wxpython.org/Phoenix/docs/html/MigrationGuide.html

The focus of this chapter is on giving you a quick introduction to **wxPython 4**.

What is a GUI?

Before you dig in to wxPython, I thought it would be good to explain what a GUI is. A graphical user interface is an interface that is drawn on screen that the user can then interact with. A user interface is made up of several common components such as:

- The main window
- Menu / toolbar
- Buttons
- Text entry
- Labels

Collectively, these are known as **widgets**. The wxPython toolkit provides dozens and dozens of widgets, including many complex custom widgets that are written in pure Python. As a developer, you will take these widgets and arrange them in a pleasing way for the user to interact with.

Let's get started by creating a "hello world" type application.

Hello World

When you create a user interface with wxPython, you will almost always need to create a wx.Frame and a `wx.Panel`. The `wx.Frame` is the window object that contains all the other widgets. It is literally a frame. The `Panel` is a bit different. It is a container as well, but it also enables the ability to tab between widgets. Without the `Panel`, tabbing will not work the way you expect. You can use Panels to group widgets as well.

Let's create an example `Frame` to start out:

```
# CR0101_hello_world.py

import wx

app = wx.App(False)
frame = wx.Frame(None, title='Hello World')
frame.Show()
app.MainLoop()
```

The first thing you will notice is that you import the wx module. This is a key import as you will need it for any of wxPython's core widgets. Next you instantiate the Application object: `wx.App`. You must have a `wx.App` instance to run anything in wxPython. However you may only have one of them at a time.

You will note that I have passed in `False` as its first argument. What this does is it prevents wxPython from catching stdout and redirecting it to a new frame that is automatically generated by wxPython. `False` is actually the default, but it is here so you know about the argument. You can play around with this as it's useful for debugging, but not something that you want to have enabled in production most of the time.

For the next step, you will create the `wx.Frame` instance. The frame has one required argument.

It is pretty standard to see the above though, but to be even more explicit you could rewrite that line to the following:

```
frame = wx.Frame(parent=None, title='Hello World')
```

As you can see, the frame requires you to pass in a parent. In this case, since this is the primary entry point to your application, you set the parent to None. We also set the title argument to a string because if you didn't, then it defaults to an empty string which is kind of boring. Next you call the frame's `Show()` method to make it visible on-screen.

Finally to get the application itself to run, you must call the app object's `MainLoop()` method. This starts the event loop so that your wxPython application can respond to the keyboard and widget events. When you run this code, you should see a window that looks like this:

Fig. 1-1: Hello World

While this code works, you will rarely write code that looks like the example above. Instead, most wxPython code that you will read and write is put into classes.

Hello with Classes

The reason that most code in wxPython is put into classes is because you will want to make your code more modular. To do that, you put the widgets related to the frame in the frame class and the widgets that are grouped into a panel in the panel class. You will find that this is true in all of Python's GUI frameworks, such as **Tkinter** or **PyQt**.

Let's take a moment and update the example above so that it uses classes:

```
# CR0102_hello_with_classes.py

import wx


class MyFrame(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title='Hello World')
        self.Show()
```

```
if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

In this example you will subclass `wx.Frame` and name your subclass MyFrame. Then you set up the frame in much the way as you did before except that the code for the frame goes into your `__init__()` method. You also need to call `self.Show()` to make the frame visible. The application creation is still at the end of the code as before. You also instantiate your new frame class here.

You aren't done with your modification yet. Python 3 recommends using `super()` when working with classes. The built-in `super()` function's primary purpose in wxPython is used for referring to the parent class without actually naming it. If you have some free time, I highly recommend you Google Raymond Hettinger's article on `super()` as it is quite helpful in understanding why it is so useful.

Anyway, let's update your code so that it uses `super()` too:

```
# CR0103_hello_with_classes_super.py

import wx


class MyFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Hello World')
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

You will see a lot of legacy code that does not use `super()`. However since this is a Python 3 book, you will use good practices and use `super()` for your examples.

Note that in Python 2, you were required to call super like this:

```
super(MyFrame, self).__init__(None, title='Hello World')
```

Let's move on and add a `Panel` class with a button to your example:

```
# CR0104_hello_with_panel.py

import wx


class MyPanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent)

        button = wx.Button(self, label='Press Me')

class MyFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Hello World')
        panel = MyPanel(self)
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

Here you add a panel that contains one widget: a button. You will notice that a panel should have a parent, which in this case is a `Frame`. You can make other widgets be the parent of a panel though. For example, you can nest panels inside of each other, or make a `wx.Notebook` into their parent. Regardless, you only want **one** panel as the sole widget for a frame. The panel will automatically expand to fill the frame as well if it is the only child widget of the frame. If you add a panel and a button to the frame without giving them a position or putting them in a sizer, then they will end up stacking up on top of each other. We will talk more about this later on in this chapter.

> Note: `wx.Panel` widgets enable tabbing between widgets on Windows. So if you want to be able to tab through the widgets in a form you have created, you are required to have a panel as their parent.

Anyway, when I ran this code, it ended up looking like this:

Fig. 1-2: Hello World with a Panel & Button

Now let's talk a little bit about event handling in wxPython.

Events

Events are what happens when the user uses your application. For example, when the user presses a button on their keyboard while your application is in focus, this will fire a `KeyEvent`. If the user clicks on a widget on your application, it will fire some kind of widget event. You can capture these events by creating an event binding. What this means is that you are creating a listener for a particular event that you want your application to react to. For example, if you have a button in your application, you probably want that button to do something when the user presses it. To actually get the button to do something, you will need to bind the button to the button press event.

Let's update the previous example so that the button actually does something:

```
# CR0105_button_event.py

import wx


class MyPanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent)
```

```
        button = wx.Button(self, label='Press Me')
        button.Bind(wx.EVT_BUTTON, self.on_button_press)

    def on_button_press(self, event):
        print('You pressed the button')

class MyFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Hello World')
        panel = MyPanel(self)
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

Here you call the button's `Bind()` method and tell it to bind to an event: `wx.EVT_BUTTON`. This is the button press event. The second argument is the function that you want to call when the button is pressed. Finally you create the event handler function, `on_button_press()`. You will notice that it takes an event argument. When you catch an event in wxPython, it will pass an event object to the function that you have bound to the event. This event object usually has information in it that identifies which widget called the function and a bunch of other information.

If you run this code, you should see it print out "**You pressed the button**" to stdout each time the button is pressed. Give it a try!

Before you continue, I want to mention that you can also bind the event like this:

```
self.Bind(wx.EVT_BUTTON, self.on_button_press, button)
```

If you do the binding this way, you are telling wxPython that you are binding the function to the `wx.Panel` instead of the `wx.Button`. This allows us to bind multiple widgets to the same event but different event handlers and then use `event.Skip()` to control which events get bubbled up the layers. In this example, the button in on the bottom layer, the panel is in the next layer up and the frame is at the top layer.

Let's update the code one more time:

```
# CR0106_event_hierarchy.py

import wx


class MyPanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent)

        button = wx.Button(self, label='Press Me')
        self.Bind(wx.EVT_BUTTON, self.panel_button_handler, button)
        button.Bind(wx.EVT_BUTTON, self.on_button_press)

    def panel_button_handler(self, event):
        print('panel_button_handler called')

    def on_button_press(self, event):
        print('on_button_press called')
        event.Skip()

class MyFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Hello World')
        panel = MyPanel(self)
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

Here you bind `EVT_BUTTON` to both the panel and the button object, but you have them call different event handlers. When you press the button, its event handler gets called immediately and it will print out the appropriate string. Then you call `event.Skip()` so that the `EVT_BUTTON` event goes up to the next event handler, if one exists. In this case, you have one for the panel and it fires as well. If you wanted to, you could also bind the frame to `EVT_BUTTON` and catch it there as well. At any of these points, you could remove the call to `event.Skip()` and the event would stop propagating at that event handler.

Absolute Positioning vs Sizers

wxPython supports both absolute positioning of widgets and relative positioning of widgets. Relative positioning of widgets requires the use of special container objects called Sizers or Layouts. A sizer allows you to resize your application and have the widgets resize along with it. If you are using absolute positioning, the widgets cannot expand or change position at all so when you resize your application, you will find that some of your widgets may get cut off. This can also happen if you load your application on a computer that is using a low resolution display. It is always recommended that you use sizers as they will make your application much nicer to use on multiple displays and screen sizes.

If you would like to play around with absolute positioning, all you need to do is update the call to the instantiation of the `wx.Button` you created earlier.

Here is an example:

```
button = wx.Button(self, label='Press Me', pos=(100, 10))
```

The new argument here is called `pos` for Position. It takes a tuple of **x** and **y** coordinates in pixels. The start location, or origin, is the top left or (0, 0). In the example above, you tell wxPython to place the button 100 pixels from the left-hand side of the panel and 10 pixels from the top.

This is what it looks like when you do that:

Fig. 1-3: Absolute Positioning

Now let's try using a sizer.

The wxPython toolkit has several sizers you can use:

- `wx.BoxSizer`
- `wx.StaticBoxSizer`
- `wx.GridSizer`
- `wx.FlexGridSizer`
- `wx.WrapSizer`

You can also nest sizers in each other. For demonstration purposes, you will focus on `wx.BoxSizer`. Let's try to center the button in our application both horizontally and vertically.

Here is an example using a slightly modified version of our previous code:

```
# CR0107_simple_sizer.py

import wx

class MyPanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent)

        button = wx.Button(self, label='Press Me')

        main_sizer = wx.BoxSizer(wx.HORIZONTAL)
        main_sizer.Add(button, proportion=0,
                       flag=wx.ALL | wx.CENTER,
                       border=5)
        self.SetSizer(main_sizer)


class MyFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Hello World')
        panel = MyPanel(self)
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

The main portion of code that we care about are these three lines:

```
main_sizer = wx.BoxSizer(wx.HORIZONTAL)
main_sizer.Add(button, proportion=0, flag=wx.ALL | wx.CENTER, border=5)
self.SetSizer(main_sizer)
```

This creates a `BoxSizer` that is Horizontally oriented, which is actually the default. Next you add the button object to our sizer and tell the sizer that the proportion of the widget should be 0, which means that the widget should be minimally sized. Then you pass in two flags: `wx.ALL` and `wx.CENTER` The first tells wxPython that you want to apply a border on all four sides of the widget while the second argument tells wxPython to center the widget. Finally you set the border to 5 pixels and since you passed in the wx.ALL flag earlier, that means you want a 5-pixel border on the top, bottom, left and right of the widget.

When I ran this, I got the following:

Fig. 1-4: Using a sizer

Interesting. The widget appears to be centered vertically in a horizontal sizer. If you look at the documentation you will find that horizontal sizers center between the bottom and the top of the parent while vertically oriented sizers align left and right.

Note: Sizers are invisible to the user, so they can be hard to visualize. You can make them appear if you use the **Widget Inspection Tool** which is helpful for debugging layouts that aren't working well. Go to https://wxpython.org/Phoenix/docs/html/wx.lib.mixins.inspection.html for an example or check out Appendix B for more information.

Let's try changing the proportion flag:

```
main_sizer.Add(button, proportion=1, flag=wx.ALL | wx.CENTER, border=5)
```

Now the proportion is 1 which tells wxPython to make that widget fill 100% of the space:

Fig. 1-5: Sizer with a proportion of 1

As you can see, the button is now stretched out horizontally across the entire application.

If you would like to make the button stretch in both directions, you can append the `wx.EXPAND` flag:

```
main_sizer.Add(button, proportion=1, flag=wx.ALL | wx.CENTER | wx.EXPAND, border=5)
```

Which will now make your application look like this on Windows and Linux:

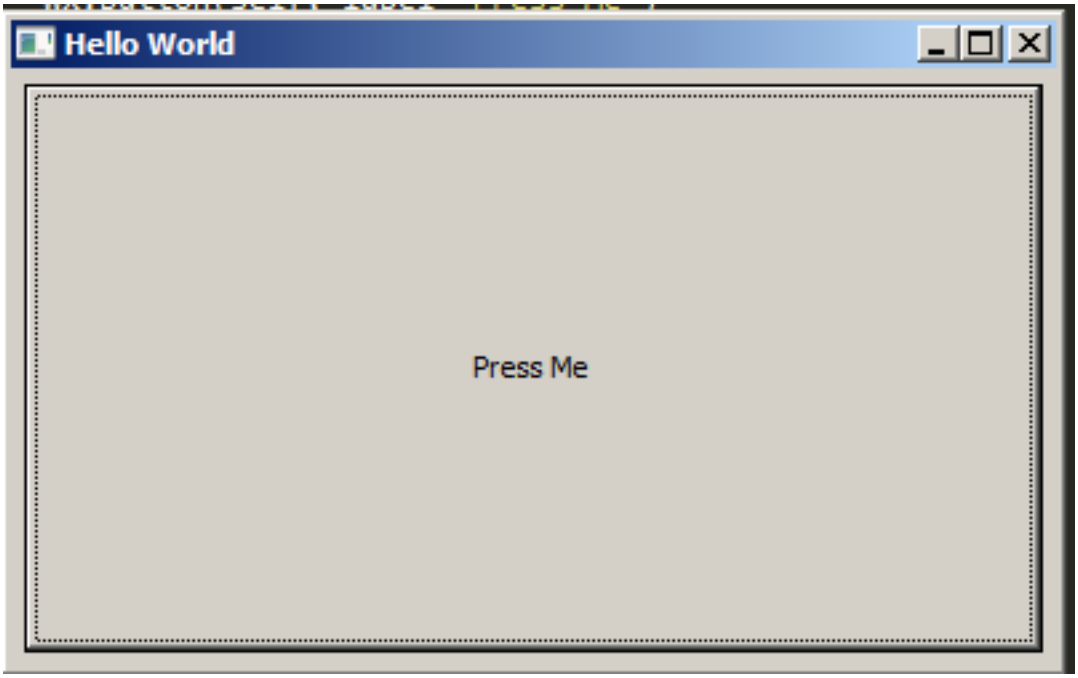

Fig. 1-6: Sizer with proportion of 1 and Expand on Windows

Note: On Mac OSX, the button retains its standard height and cannot be made higher in this manner. Instead, you would need to use a custom button or add an image to the standard button that would increase its height beyond the standard size.

If you happened to have multiple widgets in your sizer, then the proportion flag would work differently. Let's say you have two buttons and you add the first button with a `proportion` of 1 and the second button with a `proportion` of 0. This will cause the first button to take up as much space as it can while leaving the second button at its minimal size.

Here is the updated code:

```
# CR0108_sizer_with_two_widgets.py

import wx


class MyPanel(wx.Panel):

    def __init__(self, parent):
        super().__init__(parent)

        button = wx.Button(self, label='Press Me')
        button2 = wx.Button(self, label='Second button')
```

```
        main_sizer = wx.BoxSizer(wx.HORIZONTAL)
        main_sizer.Add(button, proportion=1,
                       flag=wx.ALL | wx.CENTER | wx.EXPAND,
                       border=5)
        main_sizer.Add(button2, 0, wx.ALL, 5)
        self.SetSizer(main_sizer)


class MyFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Hello World')
        panel = MyPanel(self)
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MyFrame()
    app.MainLoop()
```

And here is the result on Mac OSX:

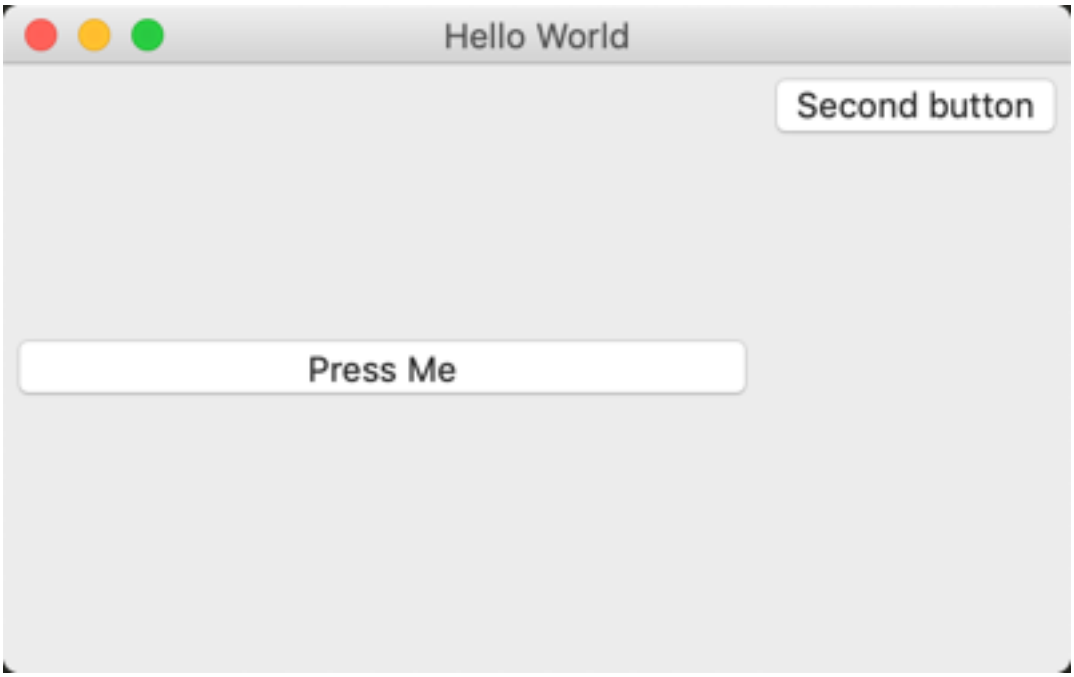

Fig. 1-7: Two buttons in a sizer

Note: The first button will be stretched out in both direction on Windows and Linux, like the screenshot in 1-6. However on Mac OSX, the height cannot be changed for `wx.Button`*. If you want the same behavior across all three platforms, you will need to use a generic button instead.*

I highly recommend playing around with the different flags and proportions using a variety of widgets. You should also check out the documentation which has lots of interesting examples in the Sizer Overview:

- https://wxpython.org/Phoenix/docs/html/sizers_overview.html

Wrapping Up

There is much, much more about wxPython that could be covered here. There are dozens upon dozens of widgets and neat features that you could talk about. However if you did that, then this chapter would end up becoming a book unto itself. This chapter is more for people to get a taste of how wxPython works so that you will be better prepared for creating actual cross-platform applications in the following chapters.

So without further ado, let's start creating!

Chapter 2 - Creating an Image Viewer

You can create pretty much anything if you put your mind to it. The biggest challenge is figuring out how to get started. Several years ago, I wanted to see how hard it would be to create a user interface that I could use to view photographs that I had taken.

Here are the two features that I required for my first version:

- Can load and display at least jpgs
- Resizes the photo so it fits on-screen

That seems really simple. In fact, I would highly recommend that when creating a proof of concept, you should always keep the number of features small. Otherwise you may spend too much time cramming unnecessary features into something that you may end up throwing out. The next step is to think about how you want your user interface to look. I find that sketching it out by hand or in software is a good way to go as it helps me see visually how the application could end up.

You can use pen and paper or you can use something like **Qt Creator** or **Visual Studio** to create really basic UIs that you would then have to code up in wxPython. I usually go for the pen and paper route, but I also have a utility called **Balsamiq Mockups** that is good for creating simple mock ups without any code or bulky applications.

I will be using that for my mock up here:

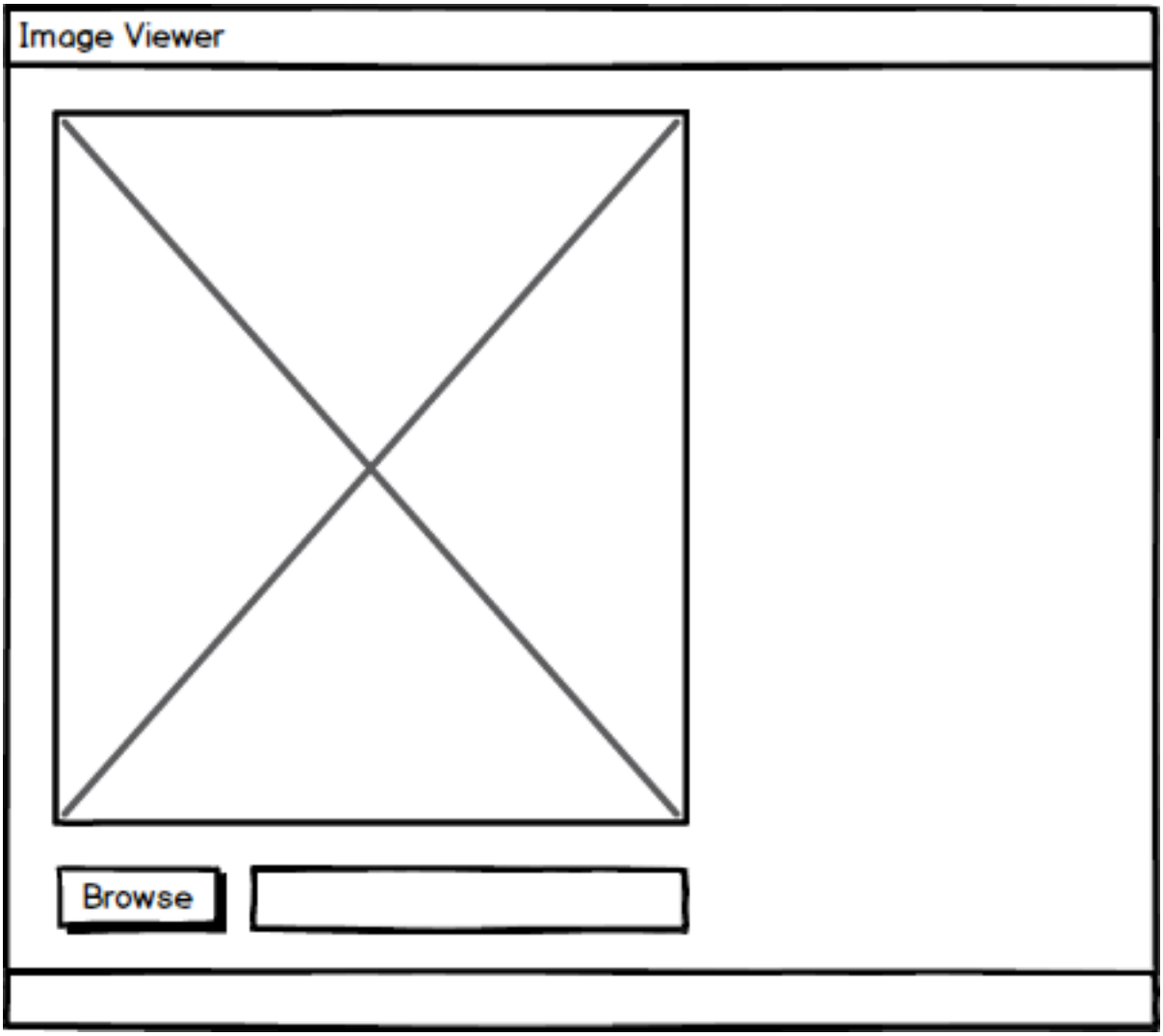

Fig. 2-1: Mockup Image Viewer

Now let's learn how to actually display a photo using wxPython. Feel free to download the code from Github if you'd like to follow along or just type the code out yourself.

Finding the Right Widget

Sometimes finding the right widget can be a challenge. Whenever I am starting out doing something completely new, I will do some research to see what is available to me. I use the following websites the most for research when it comes to wxPython:

- Google
- The wxPython documentation - > https://wxpython.org/Phoenix/docs/html/index.html
- The wxPython demo

In this case, since I know I want to display an image, I would probably look on Google and check the demo. The wxPython demo actually has a couple of candidates that I could use:

- `wx.Image`
- `wx.StaticBitmap`
- `wx.lib.agw.thumbnailctrl.ThumbnailCtrl`

The quickest method of choosing which one to actually use is to analyze what the demo itself is using. In most of the examples that I saw, it was using `wx.StaticBitmap` to display the image to the user. So we will use that as well. You will find that `wx.Image` is actually used as well for converting image files into a format that StaticBitmap can display to the user.

The `wx.Image` widget supports the following formats:

- BMP
- PNG
- JPEG
- GIF (not animated)
- PCX
- TIFF
- TGA
- IFF
- XPM
- ICO
- CUR
- ANI

For full details, see https://wxpython.org/Phoenix/docs/html/wx.Image.html

Anyway, you can use wx.Image to load your image into `wx.StaticBitmap` for display to the user. Let's get started actually coding something up.

Displaying an Image

The first task to tackle is the creation of the widget that will display an image to the user. Let's create a simple interface that has a `StaticBitmap` widget and a button.

I will focus on the `Panel` portion first as it has most of the code that you care about:

```
# image_viewer.py

import wx

class ImagePanel(wx.Panel):

    def __init__(self, parent, image_size):
        super().__init__(parent)

        img = wx.Image(*image_size)
        self.image_ctrl = wx.StaticBitmap(self,
                                          bitmap=wx.Bitmap(img))
        browse_btn = wx.Button(self, label='Browse')

        main_sizer = wx.BoxSizer(wx.VERTICAL)
        main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
        main_sizer.Add(browse_btn)
        self.SetSizer(main_sizer)
        main_sizer.Fit(parent)
        self.Layout()
```

Here you create a subclass of `wx.Panel` that you call `ImagePanel`. Next you create an instance of `wx.Image`. This is used as an initial placeholder image for when you first load up your user interface. The `wx.Image` widget accepts a width and height as its arguments. To keep things simple, you can just pass in a tuple and unpack the width and height using Python's `*` operator.

The `wx.StaticBitmap` requires a bitmap of some sort and the `wx.Image` works well for this use case. Speaking of the StaticBitmap, that is exactly what you create next. Note that you use wxPython's `wx.Bitmap` to turn your `wx.Image` instance into something that your `StaticBitmap` can use. Then you create a **Browse** button and finally you add the two widgets to your sizer. The button is not bound to any events, so it won't do anything yet if you click it.

The second to last line tells wxPython to `Fit()` the sizer to the size of the parent. This causes wxPython to attempt to match the sizer's minimal size and reduces whitespace around the widgets.

The last line calls the panel's `Layout()` method, which will force a layout of all the children widgets. It is especially useful when adding and removing widgets to a sizer or parent widget. In this case, it can be useful when working with `Fit()` and when working with image related widgets that can have their contents change.

Now let's add the following code to your Python file so that you can run your new application:

```
class MainFrame(wx.Frame):

    def __init__(self):
        super().__init__(None, title='Image Viewer')
        panel = ImagePanel(self, image_size=(240,240))
        self.Show()


if __name__ == '__main__':
    app = wx.App()
    frame = MainFrame()
    app.MainLoop()
```

This code creates a simple subclass of `wx.Frame`, instantiates your panel and shows the frame to the user. If you don't instantiate the panel class here, no widgets will be shown on-screen.

When you run this code, you should see something like the following:

Fig. 2-2: Image Viewer Skeleton

That is pretty close to your sketch except that it doesn't have the text box that should contain the path to the currently open image. Let's add that and make the button do something too!

Making Working Buttons

This next step in building your application is to make your **Browse** button do something. So let's add the text control and make the **Browse** button do something useful!

Copy the code from the previous example into a new file called **image_viewer_button_event.py** and update the `ImagePanel` class to the following:

```
# image_viewer_button_event.py

import wx

class ImagePanel(wx.Panel):

    def __init__(self, parent, image_size):
        super().__init__(parent)

        img = wx.Image(*image_size)
        self.image_ctrl = wx.StaticBitmap(self,
                                          bitmap=wx.Bitmap(img))

        browse_btn = wx.Button(self, label='Browse')
        browse_btn.Bind(wx.EVT_BUTTON, self.on_browse)

        self.photo_txt = wx.TextCtrl(self, size=(200, -1))

        main_sizer = wx.BoxSizer(wx.VERTICAL)
        hsizer = wx.BoxSizer(wx.HORIZONTAL)

        main_sizer.Add(self.image_ctrl, 0, wx.ALL, 5)
        hsizer.Add(browse_btn, 0, wx.ALL, 5)
        hsizer.Add(self.photo_txt, 0, wx.ALL, 5)
        main_sizer.Add(hsizer, 0, wx.ALL, 5)

        self.SetSizer(main_sizer)
        main_sizer.Fit(parent)
        self.Layout()
```

The piece that you should focus on here is adding an event binding to your button object.

Here is the relevant code:

```
browse_btn.Bind(wx.EVT_BUTTON, self.on_browse)
```

All this does is tell wxPython that you will now do something when the user presses the Browse button. The something that you will do is call the `on_browse()` method.

Let's write that next:

```
def on_browse(self, event):
    """
    Browse for an image file
    @param event: The event object
    """
    wildcard = "JPEG files (*.jpg)|*.jpg"
    with wx.FileDialog(None, "Choose a file",
                       wildcard=wildcard,
                       style=wx.FD_OPEN) as dialog:
        if dialog.ShowModal() == wx.ID_OK:
            self.photo_txt.SetValue(dialog.GetPath())
```

The first item to discuss is your wildcard variable. This variable holds the file types that the user is able to select when using your application. In this case, you are limiting the user to only be able to view JPEG images. The next step is to create an instance of wxPython's `wx.FileDialog`. We set its parent to `None` and give it a title, the wildcard and what style to use.

In this case, we want it to be an open file dialog, so we set the style to `wx.ID_OPEN`. Then you show the dialog to the user modally. Modal means that the dialog will appear on top of your application and prevent the user from interacting with it until they either choose a file or dismiss the dialog.

The file dialog will use the native operating system's file dialog.

On my Mac, I got this when I clicked the Browse button:

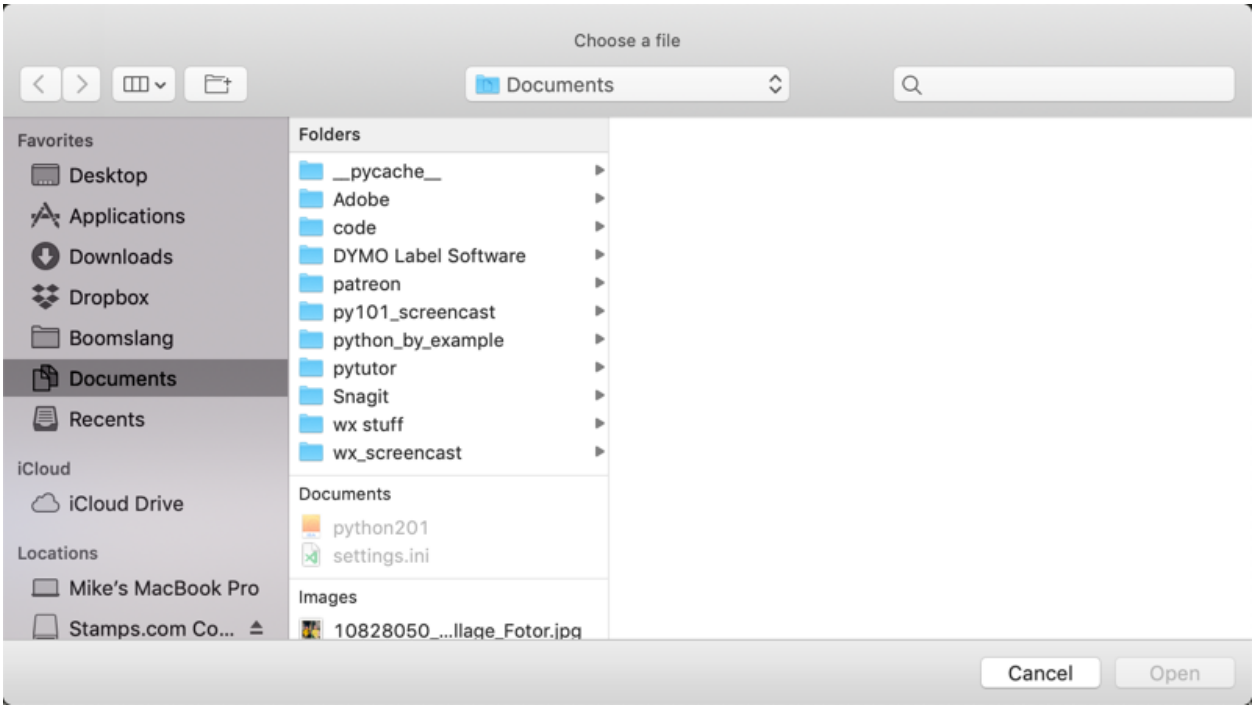

Fig. 2-3: File Browser Dialog

If you happen to run this code on Windows or Linux, the file dialog will look like that operating system's default file dialog. When the user presses OK, then the dialog will set the text control's value to the path of the file that the user selected. Python's `with` statement will automatically call the dialog's `Destroy()` method for you and prevent the dialog from hanging in your computer's memory.

Try running this code and selecting a JPEG file on your local system.

You should end up with something like this:

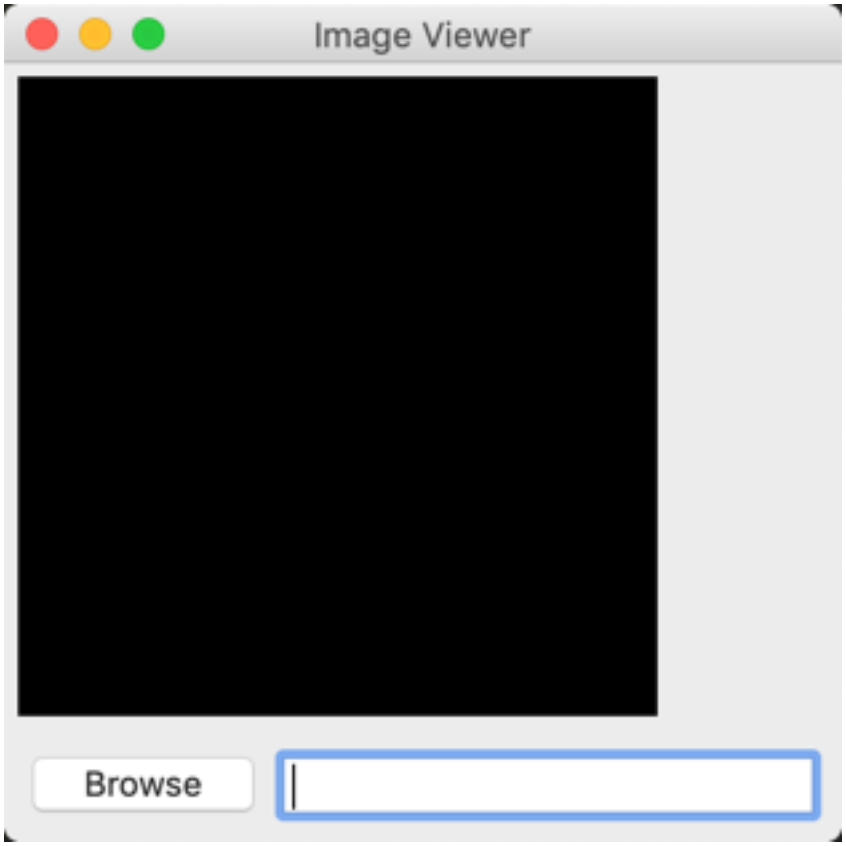

Fig. 2-4: Image Viewer Skeleton with TextCtrl

You will notice that the image still isn't being loaded in your application. Let's learn how to do that!

Loading an Image

Loading and displaying the image is actually quite easy to do with wxPython. But first you need to add an instance attribute that determines the maximum size allowed for the image. This will prevent our application from loading an image into the control that is too large to be displayed. We will add this attribute at the beginning of your `wx.Panel` subclass. Take the code from the previous section and copy and paste it into a new file named **image_viewer_working.py**.

Then update it to include a new instance attribute called `self.max_size`:

```
# image_viewer_working.py

import wx

class ImagePanel(wx.Panel):

    def __init__(self, parent, image_size):
        super().__init__(parent)
        self.max_size = 240

        # Rest of code snipped for brevity
```

Leave the rest of this method alone. The next step is to update your `on_browse()` method to call a new method:

```
def on_browse(self, event):
    """
    Browse for an image file
    @param event: The event object
    """
    wildcard = "JPEG files (*.jpg)|*.jpg"
    with wx.FileDialog(None, "Choose a file",
                       wildcard=wildcard,
                       style=wx.FD_OPEN) as dialog:
        if dialog.ShowModal() == wx.ID_OK:
            self.photo_txt.SetValue(dialog.GetPath())
            self.load_image()
```

Here you call a new method when the user presses the **OK** button in the open file dialog called `load_image()`.

This is the method you need to write next:

```
def load_image(self):
    """
    Load the image and display it to the user
    """
    filepath = self.photo_txt.GetValue()
    img = wx.Image(filepath, wx.BITMAP_TYPE_ANY)

    # scale the image, preserving the aspect ratio
    W = img.GetWidth()
    H = img.GetHeight()
```

```
    if W > H:
        NewW = self.max_size
        NewH = self.max_size * H / W
    else:
        NewH = self.max_size
        NewW = self.max_size * W / H
    img = img.Scale(NewW,NewH)

    self.image_ctrl.SetBitmap(wx.Bitmap(img))
    self.Refresh()
```

Here you grab the file path that is in your text control. Then you attempt to load that image using wxPython's `wx.Image` class and tell it to accept pretty much any of the supported file types by using the `wx.BITMAP_TYPE_ANY` flag. Of course, you currently have the file dialog itself set to only allow you to pick JPG files. But if you loosened up that restriction, you could accept other image types.

The next thing you do is some scaling to make sure that the image gets scaled to fit your max size, which is 240 pixels. You can use your image object's `Scale()` method here and pass it your calculated width and height.

Finally you use your StaticBitmap control's `SetBitmap()` method to actually display the image to the user. It requires you to pass it an instance of `wx.Bitmap`, so you use `wx.Bitmap` to create one of those on the fly and put it into your `StaticBitmap` control. Lastly you call the panel's `Refresh()` method to force a refresh.

I ran this code and used it to open up the cover image for one of my other books:

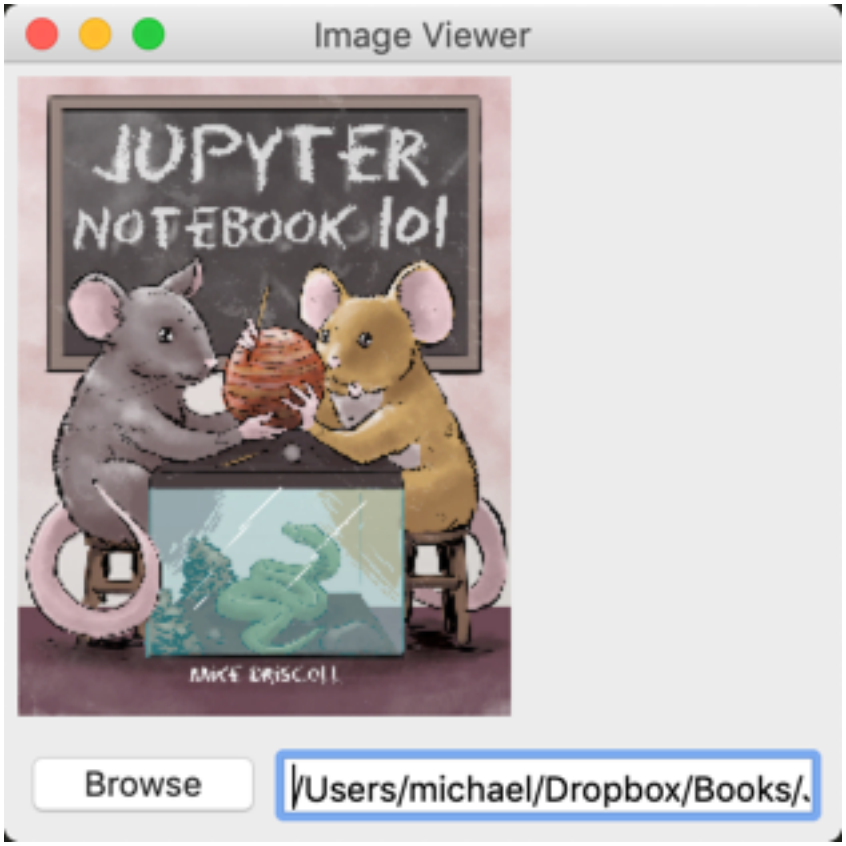

Fig. 2-5: Displaying an image

Try opening a few of your own photos to verify it works for you too!

Wrapping Up

We learned a lot about how wxPython works and how easy it is to write an application that can load and display images. The full code for this chapter ended up being only 79 lines including docstrings. I think that is quite good for a simple cross-platform application. However this photo viewer is pretty limited. It would be nice if you could load up a folder of images and then have a "previous" and "next" button to cycle through the photos in said folder. We will look into how to add that feature and another one in the next chapter!