CHAPTER 16

Threads and Timers

Recipe 16-1. How to Update a Progress Bar from a Thread

Problem

If you use GUIs (graphical user interfaces) in Python much, you know that every now and then you need to execute some long-running process. Of course, if you do that as you would with a command-line program, then you'll be in for a surprise. In most cases, you'll end up blocking your GUI's event loop and the user will see your program freeze. This is true of all the Python GUI toolkits, including Tkinter, PyQt, or wxPython. What can you do to get around such mishaps? Start the task in another thread or process, of course! In this chapter, we'll look at how to do this with wxPython and Python's threading module.

In the wxPython world, there are three related "thread-safe" methods. If you do not use one of these three when you go to update your user interface, then you may experience weird issues. Sometimes your GUI will work just fine. Other times, it will crash Python for no apparent reason, thus the need for the thread-safe methods: wx.PostEvent, wx.CallAfter, and wx.CallLater. According to Robin Dunn (creator of wxPython), wx.CallAfter uses wx.PostEvent to send an event to the application object. The application will have an event handler bound to that event and will react according to whatever the programmer has coded upon receipt of the event. It is my understanding that wx.CallLater calls wx.CallAfter with a specified time limit so that you can tell it how long to wait before sending the event.

Robin Dunn also pointed out that the Python Global Interpreter Lock (GIL) will prevent more than one thread to be executing Python bytecodes at the same time, which may limit how many CPU (central processing unit) cores are utilized by your program.

M. Driscoll, *wxPython Recipes*, https://doi.org/10.1007/978-1-4842-3237-8_16

On the flip side, he also said that "wxPython releases the GIL while making calls to wx APIs so other threads can run at that time." In other words, your mileage may vary when using threads on multicore machines. I found this discussion to be interesting and confusing.

Anyway, what this means in regard to the three wx-methods is that **wx.CallLater** is the most abstract thread-safe method with **wx.CallAfter** next and **wx.PostEvent** being the lowest level. In the following examples, you will see how to use wx.CallAfter and wx.PostEvent to update your wxPython program.

Solution for wxPython 2.8.12 and Earlier

On the wxPython mailing list, you'll see the experts telling others to use **wx.CallAfter** along with **PubSub** to communicate with their wxPython applications from another thread. I've probably even told people to do that. So in the following example, that's exactly what we're going to do. Note that this code is using the old version of **PubSub** so it will only work with **wxPython 2.8.12** or older.

```
# wxPython 2.8.12

import time
import wx

from threading import Thread
from wx.lib.pubsub import Publisher

class TestThread(Thread):
    """Test Worker Thread Class."""

    def __init__(self):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.daemon = True
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(6):
```

```
            time.sleep(10)
            wx.CallAfter(self.postTime, i)
        time.sleep(5)
        wx.CallAfter(Publisher().sendMessage, "update", "Thread finished!")

    def postTime(self, amt):
        """
        Send time to GUI
        """
        amtOfTime = (amt + 1) * 10
        Publisher().sendMessage("update", amtOfTime)

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, wx.ID_ANY, "Tutorial")

        # Add a panel so it looks the correct on all platforms
        panel = wx.Panel(self, wx.ID_ANY)
        self.displayLbl = wx.StaticText(panel,
            label="Amount of time since thread started goes here")
        self.btn = btn = wx.Button(panel, label="Start Thread")

        btn.Bind(wx.EVT_BUTTON, self.onButton)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.displayLbl, 0, wx.ALL|wx.CENTER, 5)
        sizer.Add(btn, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

        # create a pubsub receiver
        Publisher().subscribe(self.updateDisplay, "update")

    def onButton(self, event):
        """
        Runs the thread
        """
        TestThread()
        self.displayLbl.SetLabel("Thread started!")
```

```
        btn = event.GetEventObject()
        btn.Disable()

    def updateDisplay(self, msg):
        """
        Receives data from thread and updates the display
        """
        t = msg.data
        if isinstance(t, int):
            self.displayLbl.SetLabel("Time since thread started: %s
            seconds" % t)
        else:
            self.displayLbl.SetLabel("%s" % t)
            self.btn.Enable()

# Run the program
if __name__ == "__main__":
    app = wx.App(False)
    frame = MyForm().Show()
    app.MainLoop()
```

How It Works

We'll be using Python's time module to fake our long-running process. However, feel free to put something better in its place. In a real-life example, I use a thread to open Adobe Reader and send a PDF to a printer. That might not seem like anything special, but when I didn't use a thread, the print button in my application would stay stuck down while the document was sent to the printer and my GUI just hung until that was done. Even a second or two is noticeable to the user!

Anyway, let's see how this works. In our thread class (reproduced in the code that follows), we override the "run" method so it does what we want. This thread is started when we instantiate it because we have **self.start()** in its **__init__** method. In the "run" method, we loop over a range of 6, sleeping for ten seconds, in between iterations and then update our user interface using wx.CallAfter and PubSub. When the loop finishes, we send a final message to our application to let the user know what happened.

```
class TestThread(Thread):
    """Test Worker Thread Class."""

    def __init__(self):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.daemon = True
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(6):
            time.sleep(10)
            wx.CallAfter(self.postTime, i)
        time.sleep(5)
        wx.CallAfter(Publisher().sendMessage, "update", "Thread finished!")

    def postTime(self, amt):
        """
        Send time to GUI
        """
        amtOfTime = (amt + 1) * 10
        Publisher().sendMessage("update", amtOfTime)
```

Notice that in our wxPython code, we start the thread using a button event handler. We also disable the button so we don't accidentally start additional threads. That would be pretty confusing if we had a bunch of them going and the UI would randomly say that it was done when it wasn't. That is a good exercise for the reader though. You could display the PID (process ID) of the thread so you'd know which was which . . . and you might want to output this information to a scrolling text control so you can see the activity of the various threads.

The last piece of interest here is probably the PubSub receiver and its event handler.

```
def updateDisplay(self, msg):
    """
    Receives data from thread and updates the display
    """
```

```
    t = msg.data
    if isinstance(t, int):
        self.displayLbl.SetLabel("Time since thread started: %s seconds" % t)
    else:
        self.displayLbl.SetLabel("%s" % t)
        self.btn.Enable()
```

See how we extract the message from the thread and use it to update our display? We also use the type of data we receive to tell us what to show the user. Pretty cool, huh?

Solution for wxPython 3 and Newer

As you may recall from previous recipes, the PubSub module was changed in wxPython 2.9 so the code in the previous section won't work with current versions of wxPython. So let's update the code a bit to make it work for wxPython 3.0 Classic and wxPython 4.

```
# wxPython 3.0 and Newer

import time
import wx

from threading import Thread
from wx.lib.pubsub import pub

class TestThread(Thread):
    """Test Worker Thread Class."""

    def __init__(self):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(6):
            time.sleep(2)
            wx.CallAfter(self.postTime, i)
```

```
        time.sleep(5)
        wx.CallAfter(pub.sendMessage, "update", msg="Thread finished!")

    def postTime(self, amt):
        """
        Send time to GUI
        """
        amtOfTime = (amt + 1) * 10
        pub.sendMessage("update", msg=amtOfTime)

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, wx.ID_ANY, "Tutorial")

        # Add a panel so it looks the correct on all platforms
        panel = wx.Panel(self, wx.ID_ANY)
        self.displayLbl = wx.StaticText(panel,
                                        label="Amount of time since thread
                                        started goes here")
        self.btn = btn = wx.Button(panel, label="Start Thread")

        btn.Bind(wx.EVT_BUTTON, self.onButton)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.displayLbl, 0, wx.ALL|wx.CENTER, 5)
        sizer.Add(btn, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

        # create a pubsub receiver
        pub.subscribe(self.updateDisplay, "update")

    def onButton(self, event):
        """
        Runs the thread
        """
        TestThread()
        self.displayLbl.SetLabel("Thread started!")
        btn = event.GetEventObject()
        btn.Disable()
```

```
    def updateDisplay(self, msg):
        """
        Receives data from thread and updates the display
        """
        t = msg
        if isinstance(t, int):
            self.displayLbl.SetLabel("Time since thread started: %s
            seconds" % t)
        else:
            self.displayLbl.SetLabel("%s" % t)
            self.btn.Enable()

# Run the program
if __name__ == "__main__":
    app = wx.App(False)
    frame = MyForm().Show()
    app.MainLoop()
```

How It Works

Note that we just ended up importing pub and replacing all the references to Publisher() with pub. We also had to change the sendMessage call slightly in that we need to call it using keyword arguments that match the function that is called by the subscriber. They're all minor changes but necessary to get them to work in newer versions of wxPython. Now let's go down a level and check out how to do it with wx.PostEvent instead.

wx.PostEvent and Threads

The following code is based on an example from the wxPython wiki. It's a little bit more complicated than the wx.CallAfter code we just looked at, but I'm confident that we can figure it out.

```
import time
import wx

from threading import Thread
```

```
# Define notification event for thread completion
EVT_RESULT_ID = wx.NewId()

def EVT_RESULT(win, func):
    """Define Result Event."""
    win.Connect(-1, -1, EVT_RESULT_ID, func)

class ResultEvent(wx.PyEvent):
    """Simple event to carry arbitrary result data."""
    def __init__(self, data):
        """Init Result Event."""
        wx.PyEvent.__init__(self)
        self.SetEventType(EVT_RESULT_ID)
        self.data = data

class TestThread(Thread):
    """Test Worker Thread Class."""

    def __init__(self, wxObject):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.wxObject = wxObject
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(6):
            time.sleep(10)
            amtOfTime = (i + 1) * 10
            wx.PostEvent(self.wxObject, ResultEvent(amtOfTime))
        time.sleep(5)
        wx.PostEvent(self.wxObject, ResultEvent("Thread finished!"))

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, wx.ID_ANY, "Tutorial")
```

```
        # Add a panel so it looks the correct on all platforms
        panel = wx.Panel(self, wx.ID_ANY)
        self.displayLbl = wx.StaticText(panel, label="Amount of time since
        thread started goes here")
        self.btn = btn = wx.Button(panel, label="Start Thread")

        btn.Bind(wx.EVT_BUTTON, self.onButton)


        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.displayLbl, 0, wx.ALL|wx.CENTER, 5)
        sizer.Add(btn, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

        # Set up event handler for any worker thread results
        EVT_RESULT(self, self.updateDisplay)

    def onButton(self, event):
        """
        Runs the thread
        """
        TestThread(self)
        self.displayLbl.SetLabel("Thread started!")
        btn = event.GetEventObject()
        btn.Disable()

    def updateDisplay(self, msg):
        """
        Receives data from thread and updates the display
        """
        t = msg.data
        if isinstance(t, int):
            self.displayLbl.SetLabel("Time since thread started: %s
            seconds" % t)
        else:
            self.displayLbl.SetLabel("%s" % t)
            self.btn.Enable()
```

```
# Run the program
if __name__ == "__main__":
    app = wx.App(False)
    frame = MyForm().Show()
    app.MainLoop()
```

Let's break this down a bit. For me, the most confusing stuff is the first three pieces.

```
# Define notification event for thread completion
EVT_RESULT_ID = wx.NewId()

def EVT_RESULT(win, func):
    """Define Result Event."""
    win.Connect(-1, -1, EVT_RESULT_ID, func)

class ResultEvent(wx.PyEvent):
    """Simple event to carry arbitrary result data."""
    def __init__(self, data):
        """Init Result Event."""
        wx.PyEvent.__init__(self)
        self.SetEventType(EVT_RESULT_ID)
        self.data = data
```

The **EVT_RESULT_ID** is the key here. It links the thread to the **wx.PyEvent** and that weird "EVT_RESULT" function. In the wxPython code, we bind an event handler to the EVT_RESULT function. This allows us to use **wx.PostEvent** in the thread to send an event to our custom event class, **ResultEvent**. What does this do? It sends the data on to the wxPython program by emitting that custom **EVT_RESULT** that we bound to. I hope that all makes sense.

Once you've got that figured out in your head, read on. Are you ready? Good! You'll notice that our **TestThread** class is pretty much the same as before except that we're using wx.PostEvent to send our messages to the GUI instead of PubSub. The application programming interface (API) in our GUI's display updater is unchanged. We still just use the message's data property to extract the data we want. That's all there is to it!

Ideally, you now know how to use basic threading techniques in your wxPython programs. There are several other threading methods too which we didn't have a chance to cover here, such as using **wx.Yield** or **Queues**. Fortunately, the wxPython wiki covers these topics pretty well, so be sure to check out the links below if you're interested in those methods.

Recipe 16-2. How to Update a Progress Bar from a Thread

Problem

A fairly common task is the need to update a progress bar every so often. In this recipe, we will create a frame with a button. When the button is pushed, it will launch a dialog that contains our progress bar and it will start a thread. The thread is a dummy thread in that it doesn't do anything in particular except send an update back to the dialog once a second for 20 seconds. Then the dialog is destroyed.

Solution

Let's start by looking at how we can accomplish this task using wxPython 2.8.12.1 which is still a popular version of wxPython even though it's pretty old.

```
import time
import wx

from threading import Thread
from wx.lib.pubsub import Publisher

class TestThread(Thread):
    """Test Worker Thread Class."""

    def __init__(self):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.daemon = True
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(20):
```

```
            time.sleep(0.25)
            wx.CallAfter(Publisher().sendMessage, "update", "")

class MyProgressDialog(wx.Dialog):
    """"""

    def __init__(self):
        """Constructor"""
        wx.Dialog.__init__(self, None, title="Progress")
        self.count = 0

        self.progress = wx.Gauge(self, range=20)


        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.progress, 0, wx.EXPAND)
        self.SetSizer(sizer)

        # create a pubsub listener
        Publisher().subscribe(self.updateProgress, "update")

    def updateProgress(self, msg):
        """
        Update the progress bar
        """
        self.count += 1

        if self.count >= 20:
            self.EndModal(0)

        self.progress.SetValue(self.count)

class MyFrame(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Progress Bar Tutorial")

        # Add a panel so it looks the correct on all platforms
        panel = wx.Panel(self, wx.ID_ANY)
        self.btn = btn = wx.Button(panel, label="Start Thread")
        btn.Bind(wx.EVT_BUTTON, self.onButton)
```

```
        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(btn, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

    def onButton(self, event):
        """
        Runs the thread
        """
        btn = event.GetEventObject()
        btn.Disable()

        TestThread()
        dlg = MyProgressDialog()
        dlg.ShowModal()
        dlg.Destroy()

        btn.Enable()

# Run the program
if __name__ == "__main__":
    app = wx.App(False)
    frame = MyFrame()
    frame.Show()
    app.MainLoop()
```

Let's spend a few minutes breaking this down. We'll start at the bottom. The **MyFrame** class is what gets run first. When you run this script you should see something like the screen in Figure 16-1.

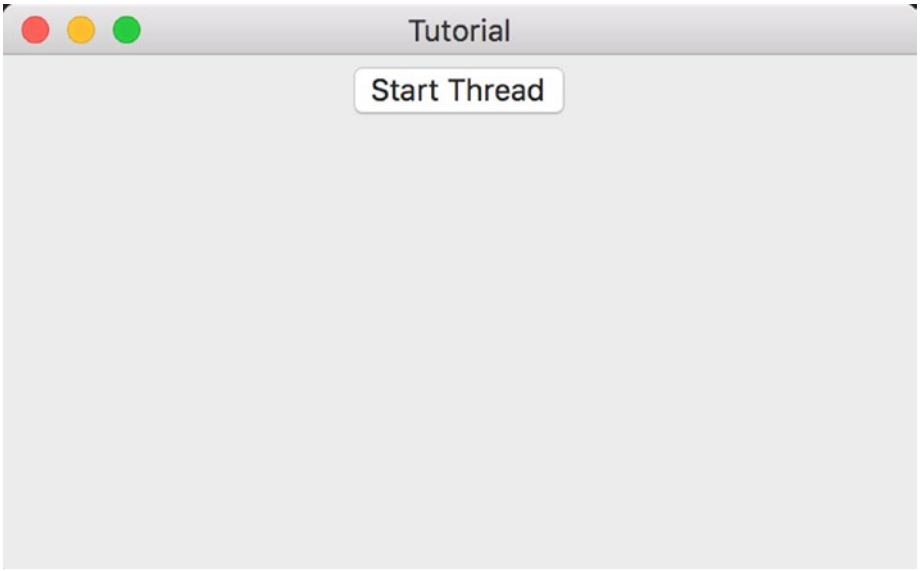

Figure 16-1. *Progress bar frame*

As you can see, all this code does is create a simple frame with a button on it. If you press the button, the following dialog will be created and a new thread will start (see Figure 16-2):

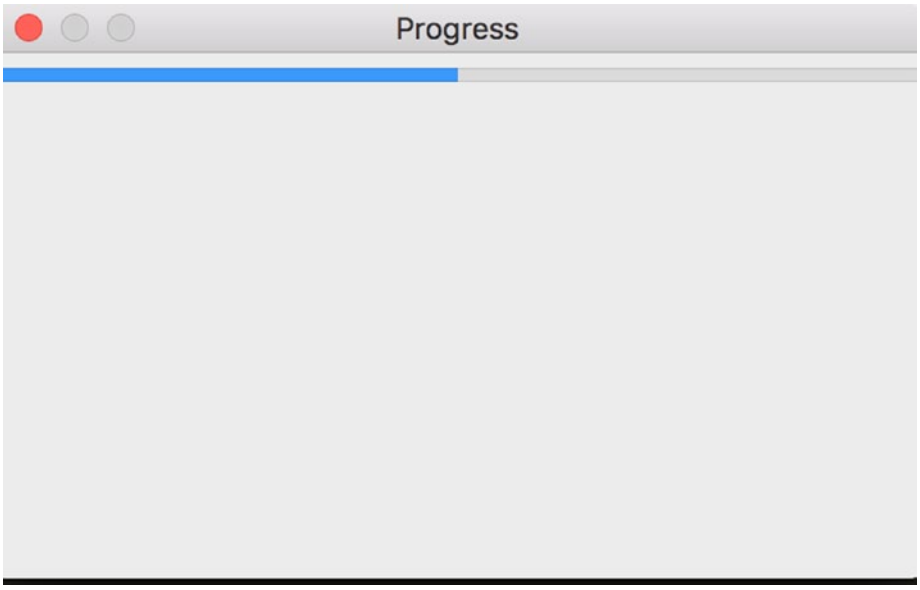

Figure 16-2. *A progress bar dialog*

Let's look at the portion of the code that makes the dialog.

```
class MyProgressDialog(wx.Dialog):
    """"""

    def __init__(self):
        """Constructor"""
        wx.Dialog.__init__(self, None, title="Progress")
        self.count = 0

        self.progress = wx.Gauge(self, range=20)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.progress, 0, wx.EXPAND)
        self.SetSizer(sizer)

        # create a pubsub listener
        Publisher().subscribe(self.updateProgress, "update")

    def updateProgress(self, msg):
        """
        Update the progress bar
        """
        self.count += 1

        if self.count >= 20:
            self.EndModal(0)

        self.progress.SetValue(self.count)
```

This code just creates a dialog with a wx.Gauge widget. The gauge is the actual widget behind the progress bar. Anyway, we create a PubSub listener at the very end of the dialog's **__init__**. This listener accepts messages that will fire off the **updateProgress** method. We will see the messages get sent in the thread class. In the updateProgress method, we increment the counter and update the wx.Gauge by setting its value. We also check to see if the count is greater than or equal to 20, which is the range of the gauge. If it is, then we close the dialog by calling its **EndModal**() method. To actually Destroy() the dialog completely, you will want to check out the frame's **onButton**() method.

Now we're ready to look at the threading code.

```
class TestThread(Thread):
    """Test Worker Thread Class."""

    def __init__(self):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(20):
            time.sleep(1)
            wx.CallAfter(Publisher().sendMessage, "update", "")
```

Here we created a thread and immediately started it. The thread loops over a range of 20 and uses the time module to sleep for a second in each iteration. After each sleep, it sends a message to the dialog to tell it to update the progress bar.

Updating the Code for wxPython 3.0.2.0 and Newer

The code in the previous section was written using PubSub's old API which has been tossed out the window with the advent of wxPython 2.9. So if you try to run the previous code in 2.9 or newer, you will likely run into issues. Thus for completeness, following is a version of the code that uses the new PubSub API and also works with wxPython Phoenix:

```
import time
import wx

from threading import Thread
from wx.lib.pubsub import pub

class TestThread(Thread):
    """Test Worker Thread Class."""
```

```
    def __init__(self):
        """Init Worker Thread Class."""
        Thread.__init__(self)
        self.daemon = True
        self.start()    # start the thread

    def run(self):
        """Run Worker Thread."""
        # This is the code executing in the new thread.
        for i in range(20):
            time.sleep(0.25)
            wx.CallAfter(pub.sendMessage, "update", msg="")

class MyProgressDialog(wx.Dialog):
    """"""

    def __init__(self):
        """Constructor"""
        wx.Dialog.__init__(self, None, title="Progress")
        self.count = 0

        self.progress = wx.Gauge(self, range=20)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.progress, 0, wx.EXPAND)
        self.SetSizer(sizer)

        # create a pubsub receiver
        pub.subscribe(self.updateProgress, "update")

    def updateProgress(self, msg):
        """"""
        self.count += 1

        if self.count >= 20:
            self.EndModal(0)

        self.progress.SetValue(self.count)
```

```
class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, wx.ID_ANY, "Tutorial")

        # Add a panel so it looks the correct on all platforms
        panel = wx.Panel(self, wx.ID_ANY)
        self.btn = btn = wx.Button(panel, label="Start Thread")
        btn.Bind(wx.EVT_BUTTON, self.onButton)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(btn, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

    def onButton(self, event):
        """
        Runs the thread
        """
        btn = event.GetEventObject()
        btn.Disable()

        TestThread()
        dlg = MyProgressDialog()
        dlg.ShowModal()
        dlg.Destroy()

        btn.Enable()

# Run the program
if __name__ == "__main__":
    app = wx.App(False)
    frame = MyForm().Show()
    app.MainLoop()
```

Note that now you import the pub module rather than the Publisher module. Also note that you have to use keyword arguments. See the PubSub documentation for additional information.

At this point, you should know how to create your own progress dialog and update it from a thread. You can use a variation of this code to create a file downloader. If you do that, you would need to check the size of the file you are downloading and download it in chunks so you can create the **wx.Gauge** with the appropriate range and update it as each chunk is downloaded. I hope this give you some ideas for how to use this widget in your own projects.

Recipe 16-3. A wx.Timer Tutorial

Problem

The wx.Timer allows the developer to execute code at specific intervals. In this chapter, I will cover several different ways to create timers. A timer object actually starts its own event loop that it controls without interfering the wxPython's main loop.

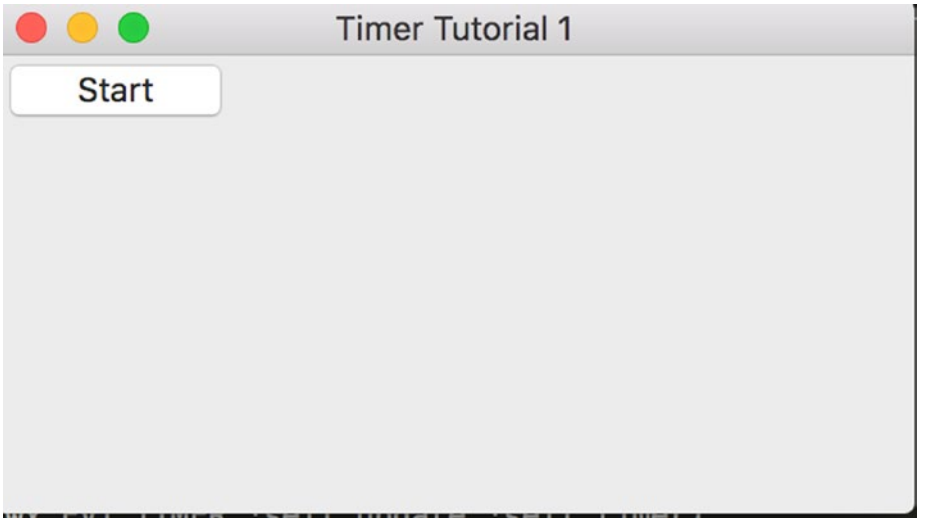

Figure 16-3. *A simple timer example*

Solution

My first example is super simple. It has only one button that starts and stops a timer. Let's take a look at the code.

```
import time
import wx

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Timer Tutorial 1",
                          size=(500,500))

        panel = wx.Panel(self, wx.ID_ANY)

        self.timer = wx.Timer(self)
        self.Bind(wx.EVT_TIMER, self.update, self.timer)

        self.toggleBtn = wx.Button(panel, wx.ID_ANY, "Start")
        self.toggleBtn.Bind(wx.EVT_BUTTON, self.onToggle)

    def onToggle(self, event):
        btnLabel = self.toggleBtn.GetLabel()
        if btnLabel == "Start":
            print("starting timer...")
            self.timer.Start(1000)
            self.toggleBtn.SetLabel("Stop")
        else:
            print("timer stopped!")
            self.timer.Stop()
            self.toggleBtn.SetLabel("Start")

    def update(self, event):
        print("\nupdated: ", time.ctime())

# Run the program
if __name__ == "__main__":
    app = wx.App(True)
    frame = MyForm().Show()
    app.MainLoop()
```

How It Works

As you can see, I only import two modules: **wx** and **time**. I use the time module to post the time that the **wx.Timer** event fires on. The two main things to pay attention to here are how to bind the timer to an event and the event handler itself. For this example to work, you have to bind the frame to the timer event. I tried binding the timer (i.e., **self.timer.Bind**), but that didn't work. So the logical thing to do was ask Robin Dunn what was going on. He said that if the parent of the timer is the frame, then the frame is the only object that will receive the timer's events unless you derive wx.Timer and override its **Notify** method. Makes sense to me.

Regardless, let's look at my event handler. In it I grab the button's label and then use a conditional **if** statement to decide if I want to start or stop the timer as well as what to label the button. In this way, I can have just one function that toggles the button and the timer's state. The part to take note of are the methods Start and Stop. They are what control the timer.

In one of my real-life applications, I have a timer execute every so often to check my e-mail. I discovered that if I shut my program down without stopping the timer, the program would basically become a zombie process. Thus, you need to make sure that you stop all your timers when your program is closed or destroyed.

Before we get to my next example, let's take a look at refactoring this one. Robin Dunn had some suggestions that I implemented in the following code. Can you tell what's different?

```
import wx
import time

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Timer Tutorial 1",
                          size=(500,500))

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

```
        self.timer = wx.Timer(self)
        self.Bind(wx.EVT_TIMER, self.update, self.timer)

        self.toggleBtn = wx.Button(panel, wx.ID_ANY, "Start")
        self.toggleBtn.Bind(wx.EVT_BUTTON, self.onToggle)

    def onToggle(self, event):
        if self.timer.IsRunning():
            self.timer.Stop()
            self.toggleBtn.SetLabel("Start")
            print("timer stopped!")
        else:
            print("starting timer...")
            self.timer.Start(1000)
            self.toggleBtn.SetLabel("Stop")

    def update(self, event):
        print("\nupdated: ", time.ctime())

# Run the program
if __name__ == "__main__":
    app = wx.App(True)
    frame = MyForm().Show()
    app.MainLoop()
```

As you can see, I've changed the event handler to check if the timer is running or not rather than looking at the button's label. This saves us one line, but it's a little cleaner and shows how to accomplish the same thing in a slightly different way.

Using Multiple Timers

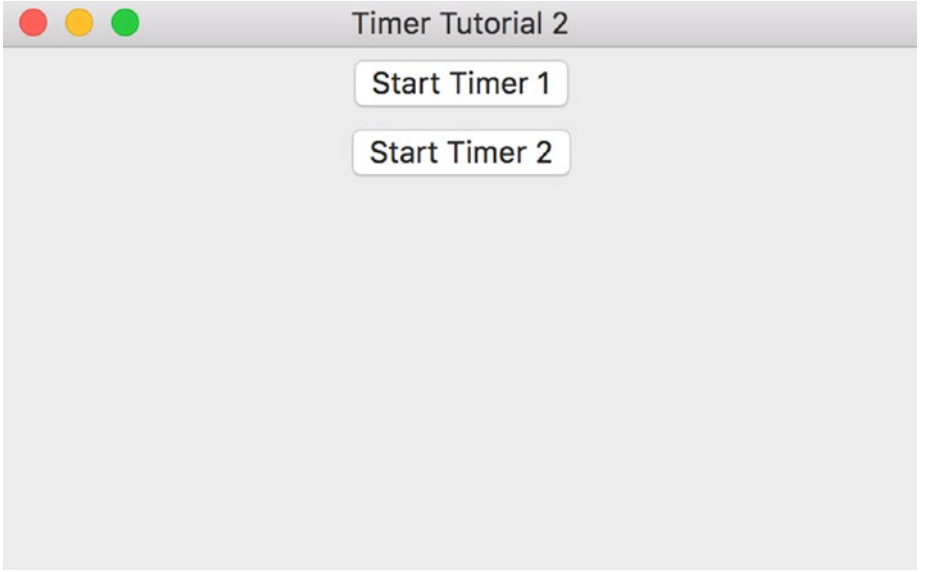

Figure 16-4. *A simple timer example*

There are many times where you will need to have multiple timers running at the same time. For example, you might need to check for updates from one or more web APIs. Here's a simple example that shows how to create a couple of timers.

```
import wx
import time

TIMER_ID1 = 2000
TIMER_ID2 = 2001

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Timer Tutorial 2")

        panel = wx.Panel(self, wx.ID_ANY)

        self.timer = wx.Timer(self, id=TIMER_ID1)
        self.Bind(wx.EVT_TIMER, self.update, self.timer)
        self.timer2 = wx.Timer(self, id=TIMER_ID2)
        self.Bind(wx.EVT_TIMER, self.update, self.timer2)
```

```
        self.toggleBtn = wx.Button(panel, wx.ID_ANY, "Start Timer 1")
        self.toggleBtn.Bind(wx.EVT_BUTTON, self.onStartTimerOne)
        self.toggleBtn2 = wx.Button(panel, wx.ID_ANY, "Start Timer 2")
        self.toggleBtn2.Bind(wx.EVT_BUTTON, self.onStartTimerOne)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.toggleBtn, 0, wx.ALL|wx.CENTER, 5)
        sizer.Add(self.toggleBtn2, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

    def onStartTimerOne(self, event):
        buttonObj = event.GetEventObject()
        btnLabel = buttonObj.GetLabel()
        timerNum = int(btnLabel[-1:])
        print(timerNum)

        if btnLabel == "Start Timer %s" % timerNum:
            if timerNum == 1:
                print("starting timer 1...")
                self.timer.Start(1000)
            else:
                print("starting timer 2...")
                self.timer2.Start(3000)
            buttonObj.SetLabel("Stop Timer %s" % timerNum)
        else:
            if timerNum == 1:
                self.timer.Stop()
                print("timer 1 stopped!")
            else:
                self.timer2.Stop()
                print("timer 2 stopped!")
            buttonObj.SetLabel("Start Timer %s" % timerNum)

    def update(self, event):
        timerId = event.GetId()
        if timerId == TIMER_ID1:
            print("\ntimer 1 updated: ", time.ctime())
```

```
        else:
            print("\ntimer 2 updated: ", time.ctime())

# Run the program
if __name__ == "__main__":
    app = wx.App()
    frame = MyForm().Show()
    app.MainLoop()
```

To be honest, this second example is mostly the same as the first one. The main difference is that I have two buttons and two timer instances. I decided to be geeky and have both buttons bind to the same event handler. This is probably one of my better tricks. To find out which button called the event, you can use the event's **GetEventObject** method. Then you can get the label off the button. If you're a real nerd, you'll notice that I could combine lines 30 and 31 into the following one-liner:

```
btnLabel = event.GetEventObject().GetLabel()
```

I split that into two lines to make it easier to follow though. Next, I used some string slicing to grab the button's label number so I would know which timer to stop or start. Then my program enters my nested **if** statements where it checks the button label and then the timer number. Now you know how to start and stop multiple timers too.

Once again, Robin Dunn came up with a better way to do this second example, so let's see what he came up with.

```
import wx
import time

class MyForm(wx.Frame):

    def __init__(self):
        wx.Frame.__init__(self, None, title="Timer Tutorial 2")

        panel = wx.Panel(self, wx.ID_ANY)

        self.timer = wx.Timer(self, wx.ID_ANY)
        self.Bind(wx.EVT_TIMER, self.update, self.timer)
        self.timer2 = wx.Timer(self, wx.ID_ANY)
        self.Bind(wx.EVT_TIMER, self.update, self.timer2)
```

```
        self.toggleBtn = wx.Button(panel, wx.ID_ANY, "Start Timer 1")
        self.toggleBtn.Bind(wx.EVT_BUTTON, self.onStartTimer)
        self.toggleBtn2 = wx.Button(panel, wx.ID_ANY, "Start Timer 2")
        self.toggleBtn2.Bind(wx.EVT_BUTTON, self.onStartTimer)

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(self.toggleBtn, 0, wx.ALL|wx.CENTER, 5)
        sizer.Add(self.toggleBtn2, 0, wx.ALL|wx.CENTER, 5)
        panel.SetSizer(sizer)

        # Each value in the following dict is formatted as follows:
        # (timerNum, timerObj, secs between timer events)
        self.objDict = {self.toggleBtn: (1, self.timer, 1000),
                        self.toggleBtn2: (2, self.timer2, 3000)}

    def onStartTimer(self, event):
        btn = event.GetEventObject()
        timerNum, timer, secs = self.objDict[btn]
        if timer.IsRunning():
            timer.Stop()
            btn.SetLabel("Start Timer %s" % timerNum)
            print("timer %s stopped!" % timerNum)
        else:
            print("starting timer %s..." % timerNum)
            timer.Start(secs)
            btn.SetLabel("Stop Timer %s" % timerNum)

    def update(self, event):
        timerId = event.GetId()
        if timerId == self.timer.GetId():
            print("\ntimer 1 updated: ", time.ctime())
        else:
            print ("\ntimer 2 updated: ", time.ctime())

# Run the program
if __name__ == "__main__":
    app = wx.App()
    frame = MyForm().Show()
    app.MainLoop()
```

In the **__init__** I added a dictionary that is keyed on the button objects. The values of the dictionary are the timer number, the timer object, and the number of seconds (technically milliseconds) between timer events. Next, I updated the button event handler to grab the button object from the event's **GetEventObject** method and then extract the respective values using said object for the dict's key. Then I can use the same trick I used in the refactored example I detailed previously, namely, the checking of whether or not the timer is running.

At this point you should have a pretty good handle on how you might use a **wx.Timer** in your own code base. It's a very easy way to fire an event at a specific time interval and it works pretty reliably. I have used timer objects in many projects. One good example was when I needed to check for updates in an e-mail alert program I had written. I used a timer to check my e-mail every so often to see if I had received anything new and to alert me if I did.