FCM155 - HowTo - Python in the REAL World - Part 103

By Greg Walters

A quick update on the recipe-scrapers library that I wrote about last month. Right now, there are 47 different recipe sites that are now supported, 5 of which I wrote for the project. I have a few more sites that I am working on for the project, so that number will continue to grow. Many of the scrapers are dedicated to non-United States sites in many different languages. I probably will explain the process of creating your own scraper in a future article.

As most of you know, I act as an unofficial support outlet for Page, the GUI designer for Python using Tkinter. On average I spend probably 5 hours a week answering questions from users, both brand new to Page and users who have been using Page for years. I also help out Don Rozenberg with testing new development builds, trying to break it in as many ways as I can. Once I can't break it anymore, Don usually creates a release. It's a very time consuming job, but one I really enjoy.

Of all the questions I get about using Page, the one I get the most is about how to create a program with multiple forms. The answer to this is really pretty easy, but is not as straight-forward as a user would think, hence the questions. The next question is how do I get the forms or windows to communicate with each other. My good friend Halvard from Norway asked how to have one form read information from another in "real time". Again, while the answer is easy, it's not something that most users would try before they ask the question. So, in this edition, I intend to present a very simple demo to help understand the process. AND there's an added benefit in that this solution is not limited to a Page GUI program. It can be used for any Python program including CLI programs. Your imagination is the only limit.

So, I thought I'd throw together a quick demo using Page to show how to deal with both ideas. I'm going to use Page since the process is much easier to show in a GUI, and since the first question is about dealing with multiple form Page programs.

The project will consist of two Page forms, one called "Parent" and the other called "Child". The Parent program will launch the Child program and will receive data from the child.

Form Design

We'll create the Parent form first. I won't bore you with the form creation details, I'll just provide a screenshot of the form and give you a small amount of important information about some special attributes of some of the widgets.

Here is what the Parent form looks like. I didn't spend much time on making it pretty and mainly kept to the attributes for everything on the form to their defaults where possible.

As you can see, it's a very simple form. Two buttons and four labels. The two buttons are named or aliased (from left to right) as **btnLaunch** and **btnExit**. There two static lables (status and received) and two dynamic labels, one that is a simple coloured square which shows the connection status to the child form and one (that shows as a fancy ribbed box) which has the textvariable set as **DataReceived** so it can be updated easily by code. This will, as the variable suggests, the data the comes from the child form. Before I saved the form, I moved it to the middle left of the screen. The child form will be positioned to the middle right side of the screen.

btnLaunch has the command attribute set to "**on_btnLaunch**" and the **btnExit** has the command attribute set to "**on_btnExit"**. These are the names of the callback functions for each of the buttons. The only other thing is that the top_level form has the title of "**I am Parent**".

Next, we'll design our child program. This one is a bit more complicated, but not horribly so.

As you can see, there is a small simple keypad similar to a “10 key” with 3 helper buttons all in a frame. There is also a label that will display the value of the key presses that has the textvariable set to **DisplayLabel**. The helper buttons are **Clear**, **Bksp** (Backspace) and **Enter**. At this point the enter key does nothing. There is also an Exit button.

The three helper buttons have the command attributes set to on_btnClear, on_btnBackspace and on_btnEnter, which are again the callback functions and the exit button has it’s command attribute set to on_btnExit.

The 11 keypad buttons don’t use the command attribute set, since it’s easier to set the callback function, which requires a parameter containing which button was clicked, by using the bind command, which we will see in a few minutes. We’ll set the bindings in the _support module

The Communications Magic

To communicate between programs, we use a shared python file, in this case called “**shared.py**”. Really apropos filename, huh? This module is imported into both (or as many programs as needed) programs as a standard import…

```
Import shared
```

The file itself is actually an empty file. There is nothing in it. However, since both of our programs have imported it, they can each read and write from and to it.

However, you need to be careful to make sure that before you try to read the value of a variable from it, the value must have already been written to the shared module. We’ll discuss this some more when we examine the code.

The Code

The code for the two _support.py modules will be presented below. It's almost presented in entirety. The code for the GUI files won't be presented, nor will the .tcl files, however the GUI.py files will be available from the pastebin repository so you can actually run the programs.

As always, we'll start with the imports section for the parent_support.py file. Notice we import child.py, child_support.py and shared.py

```
import sys
import child
import child_support
import shared
```

The next function is provided by Page for us, which is the set_Tk_var function. This gives us the access to the Label that displays the values that are generated by the child program

```
def set_Tk_var():
    global DataReceived
    DataReceived = tk.StringVar()
    DataReceived.set('')
```

Next up is the init function. This is the very last thing that gets run within the program before the GUI is shown to the user, so we run any initialization and setup tasks from this function. The top part of the function is already written for us by Page. I always provide the comment box, just to give me a "landmark" to easily find the function. We'll discuss my added code below...

```
def init(top, gui, *args, **kwargs):
    global w, top_level, root
    w = gui
    top_level = top
    root = top
    # ========================================================
    # My init code starts...
    # ========================================================
    shared.child_active = False
    shared.ReadyToRead = False
    global LblStat
    LblStat = w.Label5
    LblStat.configure(background='RED')
    global comm1
    comm1 = root.after(0, on_tick)
```

The first two lines of my added code, set two variables in the shared module. That way, when the child program starts up, the variables are already there and can be written to when needed. We also use those variables in the next function so they need to be initialized right away. If we don't, Python will throw an error.

The next thing we do is assign an alias for the status label, which is the red square that shows when the child program is running and connected to the shared module. Finally, I set up a timer function that Tkinter provides called "**root.after**". This is an event that fires every X milliseconds to take care of just about any kind of repetitive task you want to do. You can also create multiple root.after timers that run simultaneously. The basic syntax is:

```
handle = root.after(ms, callback)
```

In the case above, the handle is called "**comm1**" which lets me know that this particular timer is used to communicate with the child process. Notice that I set the time to 0, which means that the callback function will be called immediately and the final parameter is the name of the callback function.

Here is the callback function code…

```
def on_tick():
    global comm1, LblStat
    if shared.child_active == True:
        LblStat.configure(background='GREEN')
        if shared.ReadyToRead:
            DataReceived.set(shared.ChildData)
            shared.ReadyToRead = False
    elif shared.child_active == False:
        LblStat.configure(background='RED')
    comm1 = root.after(100, on_tick)
```

First, we set two global variables, **comm1** which is the handle for the timer and **LblStat**, which is the alias for our little red square. Next, we access the shared module to see if the child process is running by checking **shared.child_active** to see if it's True. This is set as soon as the child program starts up. If it is, we set the square coloured label to "Green" to show that the child process is running and then we check to see if **shared.ReadyToRead** is set to True, which is basically a flag that says that one of the numeric keys has been clicked. If it is, we get that data, put it into the display label with the **.set()** method and clear the **ReadyToRead** flag, so we can wait for the next click event on the child process. If the **shared.child_active** flag is False, we reset the coloured square to Red. This way, when the child program is exited, we will know it visually.

Finally, we “re-arm” the timer routine, this time to check 100 ms from that point.

Now, we look at the on_btnExit callback function. It’s very simple. We simply call the **destroy_window()** function that will cleanly end the program. The destroy_window function is provided by Page as is the **on_btnExit** callback skeleton, since we added the callback name in the command attribute for the exit button. All we have to do is add the line “**destroy_window()**”.

```
def on_btnExit():
    # print('parent_support.on_btnExit')
    # sys.stdout.flush()
    destroy_window()
```

Next, we’ll look at the **btnLaunch** callback function. This is how we make the child program start. Again, we used the command attribute for the button in Page, so the skeleton is started for us…

```
def on_btnLaunch():
    # print('parent_support.on_btnLaunch')
    # sys.stdout.flush()
    child.create_Toplevel1(root)
```

The only line we need to enter here is the last one. Since we have already imported the child.py GUI file at the top of the code, we just need to call the **create_Toplevel1()** function. This is the entry point for the program when it is called from another program.

Finally, I’ve provided the **destroy_window()** which is provided by Page, just so you can see it.

```
def destroy_window():
    # Function which closes the window.
    global top_level
    top_level.destroy()
    top_level = None
```

Now we’ll look at the child program. It is a bit more complicated, but not overly so. Again, I’m only going to provide the code here for the **child_support.py** module.

Again, we’ll start with the import section. Notice here, that we only need to import the shared module, since we don’t need to call any functions from the parent.

```
import sys
```

```
import shared
```

Here's the definition for the label that shows the running value from the keypad button entries.

```
def set_Tk_var():
    global DisplayLabel
    DisplayLabel = tk.StringVar()
    DisplayLabel.set('Label')
```

Here is the **init** function from the child program. Again, you can see where my code starts.

```
def init(top, gui, *args, **kwargs):
    global w, top_level, root
    w = gui
    top_level = top
    root = top
    # ========================================================
    # My init code starts...
    # ========================================================
    global valu
    valu = ''
    setup_bindings()
    shared.child_active = True
```

The first thing we do is set up a global variable to hold the accumulated value of the keypad entries. We then call the function **setup_bindings()** that attach the callback function to all of the buttons of the keypad. Finally we set the **shared.child_active** flag to True.

Since we are going to pass parameters to the callback for the keypad buttons. It's much easier to deal with things here than to try to do it within Page through the command attribute.

```
def setup_bindings():
    w.btn0.bind('<Button-1>', lambda e: on_btnClick(e, 0))
    w.btn1.bind('<Button-1>', lambda e: on_btnClick(e, 1))
    w.btn2.bind('<Button-1>', lambda e: on_btnClick(e, 2))
    w.btn3.bind('<Button-1>', lambda e: on_btnClick(e, 3))
    w.btn4.bind('<Button-1>', lambda e: on_btnClick(e, 4))
    w.btn5.bind('<Button-1>', lambda e: on_btnClick(e, 5))
    w.btn6.bind('<Button-1>', lambda e: on_btnClick(e, 6))
    w.btn7.bind('<Button-1>', lambda e: on_btnClick(e, 7))
    w.btn8.bind('<Button-1>', lambda e: on_btnClick(e, 8))
    w.btn9.bind('<Button-1>', lambda e: on_btnClick(e, 9))
    w.btnDot.bind('<Button-1>', lambda e: on_btnClick(e, 10))
```

Now we define the callback routine code for when a keypad button is clicked. Notice this must be done from scratch, since Page has no idea of the need for the function.

```
def on_btnClick(e, which):
    global valu
    if which < 10:
        valu = valu + str(which)
    elif which == 10:
        valu = valu + "."
    shared.ChildData = valu
    DisplayLabel.set(valu)
    shared.ReadyToRead = True
```

In the callback, we simply take the value of which (which is the number of the button) and append it, as a string, to the valu variable. We also check to see if the period key (value 10) was pressed and if it was, then we add the period into the display value. Finally, we put the data into the **DisplayLabel** through the **.set()** method and set **shared.ReadyToRead** to True, so the parent knows to pull the data.

We don't do anything with the Enter button, so we just leave the skeleton for later use.

```
def on_btnEnter():
    print('child_support.on_btnEnter')
    sys.stdout.flush()
```

The callback for the Exit button is mostly the same as for the parent, but we also set **shared.child_active** to False, so the Parent knows to change the status square from Green to Red and not to try to poll.

```
def on_btnExit():
    print('child_support.on_btnExit')
    sys.stdout.flush()
    shared.child_active = False
    destroy_window()
```

The **btnClear** callback function sets the global **valu** to an empty string, sets data into the shared module and the display label and then sets the **shared.ReadyToRead** flag to True.

```
def on_btnClear():
    # print('child_support.on_btnClear')
    # sys.stdout.flush()
    global valu
    valu = ''
    shared.ChildData = valu
    DisplayLabel.set(valu)
    shared.ReadyToRead = True
```

Finally, we deal with the callback for the Backspace button. We simply strip the last character from the **valu** string, displays it, passes it to the shared module and sets the flag so the parent program will read it.

```
def on_btnBackspace():
    # print('child_support.on_btnBackspace')
    # sys.stdout.flush()
    global valu
    valu = valu[:len(valu)-1]
    shared.ChildData = valu
    DisplayLabel.set(valu)
    shared.ReadyToRead = True
```

As normal I've put the code for the programs on Pastebin. You can find the links below:

parent.py - https://pastebin.com/AZXXvuAU
parent_support.py - https://pastebin.com/3iBHgCN0
child.py - https://pastebin.com/bwZLnkHc
child_support.py - https://pastebin.com/Vg0K1w5G

I hope this article has given you some food for thought that can be used in your own programming.

Until next time, keep coding

Greg