User Interfaces for Humans™

The PySimpleGUI Cookbook

Welcome to the PySimpleGUI Cookbook! It's provided as but one component of a larger documentation effort for the PySimpleGUI package. Its purpose is to give you a jump start.

You'll find that starting with a Recipe will give you a big jump-start on creating your custom GUI. Copy and paste one of these Recipes and modify it to match your requirements. Study them to get an idea of some design patterns to follow.

This document is not a replacement for the main documentation at http://www.PySimpleGUI.org. If you're looking for answers, they're most likely there in the detailed explanations and the detailed call reference. That document is updated much more frequently than this one.

See the main doc on installation. Typically it's `pip install pysimplegui` to install.

Constantly Being Updated

Because PySimpleGUI is an active project, new capabilities are being added frequently, and the recommended method for doing operations evolves over time, that means this Cookbook also changes over time. However, the speed the Cookbook gets updated will, by definition, lag behind the code changes.

The Demo Programs are going to be the most up to date examples for you, but even those get out of date. It's an imperfect world, but let's make the most of what we've got.

The "Demo Programs" Are Also "Recipes"

If you like this Cookbook, then you'll LOVE the 300 sample programs that are just like these. You'll find them in the GitHub at http://Demos.PySimpleGUI.com. They are located in the folder `DemoPrograms` and there is also a `Demo Programs` folder for each of the PySimpleGUI ports.

These programs are updated frequently, much more so than this Cookbook document. It's there that you'll find the largest potential for a big jump-start on your project.

These short Demo Programs fall into 3 categories:

1. Demonstrate a particular Element
2. Integration with another package (e.g. Matplotlib, OpenCV, etc)
3. Provide added functionality - more complex element combinations or extensions to elements

So, for example, if you're trying to use the Graph Element to create a line graph, check out the demo programs... there are 8 different demos for the Graph Element alone.

Because there are so many Demo Programs, there is a "Demo Program Browser". There is a Recipe in this Cookbook on how to download the Demo Programs and run the Demo Program Browser.

Trinket, the Online PySimpleGUI Cookbook

More and more the recipes are moving online, away from this document and onto Trinket. http://Trinket.PySimpleGUI.org

You'll find a number of "recipes" running on Trinket. The PySimpleGUI Trinket Demo Programs are often accompanied by explanatory text. Because it's an actively used educational capability, you'll find newer PySimpleGUI features demonstrated there.

The advantage to "live", online PySimpleGUI demos is that you can examine the source code, run it, and see the GUI in your browser window, without installing *anything* on your local machine. No Python, no PySimpleGUI, only your browser is needed to get going.

Repl.it... another online resource

The PySimpleGUI repl.it repository is also used, but it doesn't provide the same kind of capability to provide some explanatory text and screenshots with the examples. It does, however, automatically install the latest version of PySimpleGUI for many of the examples. It also enables the demo programs to access any package that can be pip installed. Trinket does not have this more expansive capability. Some older demos are located there. You can run PySimpleGUIWeb demos using Repl.it.

Cookbook Purpose

A quick explanation about this document. The PySimpleGUI Cookbook is meant to get you started quickly. But that's only part of the purpose. The other, probably most important one, is *coding conventions*. The more of these examples and the programs you see in the Demo Programs section on the GitHub, the more familiar certain patterns will emerge.

It's through the Cookbook and the Demo Programs that new PySimpleGUI constructs and naming conventions are "rolled out" to the user community. If you are brand new to PySimpleGUI, then you're getting your foundation here. That foundation changes over time as the package improves. The old code still runs, but as more features are developed and better practices are discovered, you'll want to be using newer examples and coding conventions.

PEP8 names are a really good example. Previously many of the method names for the Elements were done with CamelCase which is not a PEP8 compliant way of naming those functions. They should have been snake_case. Now that a complete set of PEP8 bindings is available, the method names are being changed here, in the primary documentation and in the demo programs. `window.Read()` became `window.read()`. It's better that you see examples using the newer `windows.read()` names.

In short, it's brainwashing you to program PySimpleGUI a certain way. The effect is that one person has no problem picking up the code from another PySimpleGUI programmer and recognizing it. If you stick with variable names shown here, like many other PySimpleGUI users have, then you'll understand other people's code (and the demos too) quicker. So far, the coding conventions have been used by most all users. It's improved efficiency for everyone.

Keys

Keys are an extremely important concept for you to understand. They are the labels/tags/names/identifiers you give Elements. They are a way for you to communicate about a specific element with the PySimpleGUI API calls.

Keys are used to:

- inform you when one of them generates an event
- change an element's value or settings

- communicate their value when performing a `window.read()`

Important - while they are shown as strings in many examples, they can be ANYTHING (ints, tuples, objects). Anything **EXCEPT Lists**. Lists are not valid Keys because in Python lists are not hashable and thus cannot be used as keys in dictionaries. Tuples, however, can.

Keys are specified when you create an element using the `key` keyword parameter. They are used to "find elements" so that you can perform actions on them.

GETTING STARTED - Copy these design patterns!

All of your PySimpleGUI programs will utilize one of these 2 design patterns depending on the type of window you're implementing. The two types of windows are:

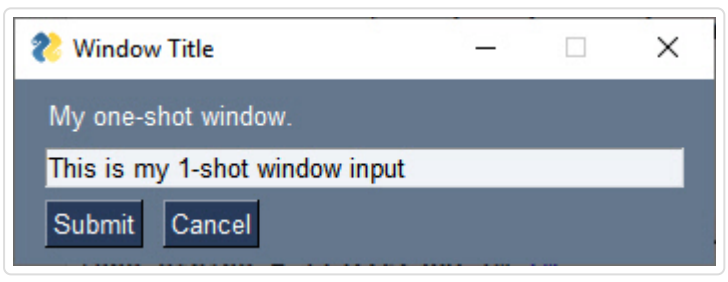
1. One-shot
2. Persistent

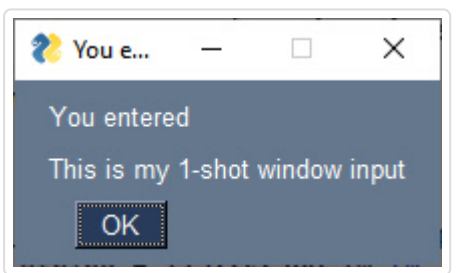
The **One-shot window** is one that pops up, collects some data, and then disappears. It is more or less a 'form' meant to quickly grab some information and then be closed.

The **Persistent window** is one that sticks around. With these programs, you loop, reading and processing "events" such as button clicks. It's more like a typical Windows/Mac/Linux program.

If you are writing a "typical Windows program" where the window stays open while you collect multiple button clicks and input values, then you'll want Recipe Pattern 2B.

Recipe - Pattern 1A - "One-shot Window" - (The Simplest Pattern)





This will be the most common pattern you'll follow if you are not using an "event loop" (not reading the window multiple times). The window is read and then closed.

When you "read" a window, you are returned a tuple consisting of an `event` and a dictionary of `values`.

The `event` is what caused the read to return. It could be a button press, some text clicked, a list item chosen, etc, or `WIN_CLOSED` if the user closes the window using the X.

The `values` is a dictionary of values of all the input-style elements. Dictionaries use keys to define entries. If your elements do not specificy a key, one is provided for you. These auto-numbered keys are ints starting at zero.

This design pattern does not specify a `key` for the `InputText` element, so its key will be auto-numbered and is zero in this case. Thus the design pattern can get the value of whatever was input by referencing `values[0]`

```
import PySimpleGUI as sg

layout = [[sg.Text('My one-shot window.')],
                [sg.InputText()],
                [sg.Submit(), sg.Cancel()]]

window = sg.Window('Window Title', layout)

event, values = window.read()
window.close()

text_input = values[0]
sg.popup('You entered', text_input)
```

If you want to use a key instead of an auto-generated key:

```
import PySimpleGUI as sg

layout = [[sg.Text('My one-shot window.')],
                [sg.InputText(key='-IN-')],
                [sg.Submit(), sg.Cancel()]]

window = sg.Window('Window Title', layout)

event, values = window.read()
window.close()

text_input = values['-IN-']
sg.popup('You entered', text_input)
```

Recipe - Pattern 1B - "One-shot Window" - (Self-closing, single line)

For a much more compact window, it's possible to create, display, read, and close a window in a single line of code.

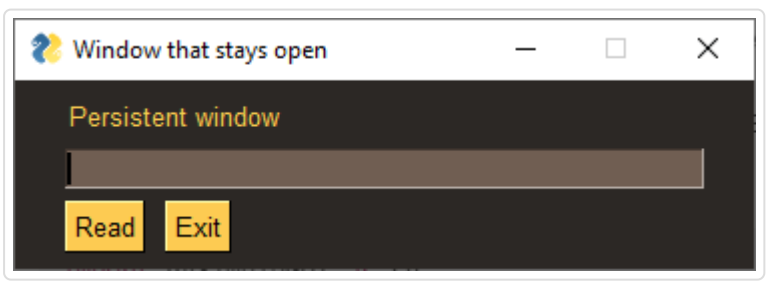
```
import PySimpleGUI as sg

event, values = sg.Window('Login Window',
                  [[sg.T('Enter your Login ID'), sg.In(key='-ID-')],
                  [sg.B('OK'), sg.B('Cancel') ]]).read(close=True)

login_id = values['-ID-']
```

The important part of this bit of code is `close=True`. This is the parameter that instructs PySimpleGUI to close the window just before the read returns.

This is a single line of code, broken up to make reading the window layout easier. It will display a window, let the user enter a value, click a button and then the window will close and execution will be returned to you with the variables `event` and `values` being returned.

Notice use of Element name "Shortcuts" (uses `B` rather than `Button`, `T` instead of `Text`, `In` rather than `InputText`, etc.). These shortcuts are fantastic to use when you have complex layouts. Being able to "see" your entire window's definition on a single screen of code has huge benefits. It's another tool to help you achieve simple code.

Recipe - Pattern 2A - Persistent window (multiple reads using an event loop)



The more advanced/typical GUI programs operate with the window remaining visible on the screen. Input values are collected, but rather than closing the window, it is kept visible acting as a way to both input and output information. In other words, a typical Window, Mac or Linux window.

Let this sink in for a moment.... in 10 lines of Python code, you can display and interact with your own custom GUI window. You are writing "*real GUI code*" (as one user put it) that will look and act like other windows you're used to using daily.

This code will present a window and will print values until the user clicks the exit button or closes window using an X.

```
import PySimpleGUI as sg

sg.theme('DarkAmber')    # Keep things interesting for your users

layout = [[sg.Text('Persistent window')],
          [sg.Input(key='-IN-')],
          [sg.Button('Read'), sg.Exit()]]

window = sg.Window('Window that stays open', layout)

while True:                             # The Event Loop
    event, values = window.read()
    print(event, values)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break

window.close()
```

Here is some sample output from this code:

```
Read {'-IN-': 'typed into input field'}
Read {'-IN-': 'More typing'}
Exit {'-IN-': 'clicking the exit button this time'}
```

The first thing printed is the "event" which in this program is the buttons. The next thing printed is the `values` variable that holds the dictionary of return values from the read. This dictionary has only 1 entry. The "key" for the entry is `'-IN-'` and matches the key passed into the `Input` element creation on this line of code:

```
          [sg.Input(key='-IN-')],
```

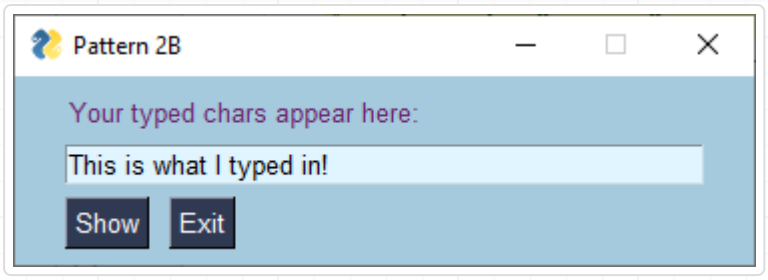
If the window was close using the X, then the output of the code will be:

```
None {'-IN-': None}
```

The `event` returned from the read is set to `None` (the variable `WIN_CLOSED`) and so are the input fields in the window. This `None` event is super-important to check for. It must be detected in your windows or else you'll be trying to work with a window that's been destroyed and your code will crash. This is why you will find this check after ***every*** `window.read()` call you'll find in sample PySimpleGUI code.

In some cirsumstances when a window is closed with an X, both of the return values from `window.read()` will be `None`. This is why it's important to check for `event is None` before attempting to access anything in the `values` variable.

Recipe - Pattern 2B - Persistent window (multiple reads using an event loop + updates data in window)



This is a slightly more complex, but more realistic version that reads input from the user and displays that input as text in the window. Your program is likely to be doing both of those activities so this pattern will likely be your starting point.

Do not worry yet what all of these statements mean. Just copy the template so you can start to experiment and discover how PySimpleGUI programs work.

```
import PySimpleGUI as sg

sg.theme('BluePurple')

layout = [[sg.Text('Your typed chars appear here:'), sg.Text(size=(15,1), key='-OUTPUT-')],
          [sg.Input(key='-IN-')],
          [sg.Button('Show'), sg.Button('Exit')]]

window = sg.Window('Pattern 2B', layout)

while True:  # Event Loop
    event, values = window.read()
    print(event, values)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == 'Show':
        # Update the "output" text element to be the value of "input" element
        window['-OUTPUT-'].update(values['-IN-'])

window.close()
```

To modify an Element in a window, you call its `update` method. This is done in 2 steps. First you lookup the element, then you call that element's `update` method.

The way we're achieving output here is by changing a Text Element with this statement:

```
window['-OUTPUT-'].update(values['-IN-'])
```

`window['-OUTPUT-']` returns the element that has the key `'-OUTPUT-'`. Then the `update` method for that element is called so that the value of the Text Element is modified. Be sure you have supplied a `size` that is large enough to display your output. If the size is too small, the output will be truncated.

There are **two important concepts when updating elements!**

1. If you need to interact with elements prior to calling `window.read()` you will need to "finalize" your window first using the `finalize` parameter when you create your `Window`. "Interacting" means calling that element's methods such as `update`, `expand`, `draw_line`, etc.
2. Your change ***will not be visible in the window until*** you either:
   A. Call `window.read()` again
   B. Call `window.refresh()`

Inside your event loop

For persistent windows, after creating the window, you have an event loop that runs until you exit the window. Inside this loop you will read values that are returned from reading the window and you'll operate on elements in your window. To operate on elements, you look them up and call their method functions such as `update`.

Old Style Element Lookups - FindElement

The original / old-style way of looking up elements using their key was to call `window.FindElement` or the shortened `window.Element`, passing in the element's key.

These 3 lines of code do the same thing. The first line is the currently accepted way of performing this lookup operation and what you'll find in all of the current demos.

```
window['-OUTPUT-']
window.FindElement('-OUTPUT-')
window.find_element('-OUTPUT-')
window.Element('-OUTPUT-')
```

Element Operations

Once you lookup an element, the most often performed operation is `update`. There are other element methods you can call such as `set_tooltip()`. You'll find the list of operations available for each element in the call summary at the end of the main documentation.

To call any of these other methods, you do the element lookup, then add on the call such as this call to `set_tooltip`.

```
window[my_key].set_tooltip('New Tooltip')
```

NOTE!

Operations on elements will not appear in your window immediately. If you wish for them to appear immediately, prior to your next `window.read()` call, you must call `window.refresh()`. A call to `read` or `refresh` causes your changes to be displayed.

Exiting a Window

For persistent windows, you will find this if statement immediately following every `window.read` call you'll find in this document and likely all of the demo programs:

```
if event in (sg.WIN_CLOSED, 'Quit'):
    break
```

or this version which is easier for beginners to understand. They perfect exactly the same check.

```
if event == sg.WIN_CLOSED or event == 'Quit':
    break
```

This is your user's "way out". **Always** give a way out to your user or else they will be using task manager or something else, all the while cursing you.

Beginners to Python may not understand this statement and it's important to understand it so that you don't simply ignore it because you don't understand the syntax.

The if statement is identical to this if statement:

```
if event == sg.WIN_CLOSED or event == 'Quit':
    break
```

The `event in (sg.WIN_CLOSED, 'Quit')` simply means is the value of the `event` variable in the list of choices shown, in this case `WIN_CLOSED` or `Quit`. If so, then break out of the Event Loop and likely exit the program when that happens for simple programs.

You may find 'Exit' instead of 'Quit' in some programs. Or may find only `WIN_CLOSED` is checked. Exit & Quit in this case refer to a Quit/Exit button being clicked. If your program doesn't have one, then you don't need to include it.

Close Your Windows

When you're done with your window, close it.

```
window.close()
```

The reason is that for some ports, like PySimpleGUIWeb, you cannot exit the program unless the window is closed. It's nice to clean up after yourself too.

Coding Conventions

By following some simple coding conventions you'll be able to copy / paste demo program code into your code with minimal or no modifications. Your code will be understandable by other PySimpleGUI programmers as well.

The primary *suggested* conventions are:

- `import PySimpleGUI as sg`
- Name your Window `window`
- Name the return values from reading your window `event` and `values`
- Name your layout `layout`
- Use `window[key]` to lookup elements
- For keys that are strings, follow this pattern `'-KEY-'`

Of course you don't have to follow *any* of these. They're suggestions, but if you do follow them, your code is a lot easier to understand by someone else.

Coding Tips

A few tips that have worked well for others. In the same spirit as the coding conventions, these a few observations that may speed up your development or make it easier for others to understand your code. They're guidelines / tips / suggestions / ideas... meant to help you.

- Stay ***simple*** at every opportunity
- Read or search the documentation (http://www.PySimpleGUI.org)
- Use the coding conventions outlined above
- Write compact layouts
- Use "user defined elements" when you find yourself repeating parameters many times (functions that return elements)
- Use PySimpleGUI constructs rather than other GUI frameworks' constructs
- Use reasonable timeout values (as large of non-zero values as possible... be a good CPU citizen)
- Do not try to make any PySimpleGUI call from a thread
- Close your windows before exiting
- Make linear event loops
- Use the `values` dictionary rather than `Element.get` methods
- Look through Demo Programs for more tips / techniques (http://Demos.PySimpleGUI.org)

Most of these are self-explanatory or will be understood as you learn more about PySimpleGUI. You won't now what a timeout value is at this point, but if/when you do use reads with timeouts, then you'll understand the tip.

A little more detail on a few of them that aren't obvious.

Write compact layouts

Try to keep your layout definitions to a single screen of code. Don't put every parameter on a new line. Don't add tons of whitespace.

If you've got a lot of elements, use the shortcut names (e.g. using `sg.B` rather than `sg.Button` saves 5 characters per button in your layout).

The idea here to be able to see your entire window in your code without having to scroll.

Use PySimpleGUI constructs

PySimpleGUI programs were not designed using the same OOP design as the other Python GUI frameworks. Trying to force fit them into an OOP design doesn't buy anything other then lots of `self.` scattered in your code, more complexity, and possibly more confusion

Of course your overall design can be OOP.

The point is that there is no concept of an "App" or a never-ending event loop or callback functions. PySimpleGUI is different than tkinter and Qt. Trying to code in that style is likely to not result in success. If you're writing a subclass for `Window` as a starting point, it's highly likely you're doing something wrong.

Recipe - The Demo Browser

There are so many demo programs that a way of quickly searching them was needed. There is the "Demo Programs Browser" that makes finding, editing and running Demo Programs easier.

Demo Program & Project Browser Features

The "PySimpleGUI Demo Program & Project Browser" makes searching for and searching inside of the PySimpleGUI Demo Programs easier. You can also use this program with any folder of Python programs. They don't have to be PySimpleGUI programs or PySimpleGUI releated.

Some of the features this program provides are: * Displays your project tree as a single list of files * "Filtering" of the files - searches by filename * "Find" files - searches inside of your list of files * Opening files using the editor of your choice * Executing the files * Easy editing of the Browser Program itself using the "Edit Me" button

Downloading the PySimpleGUI Demo Programs

Jump Start! Get the Demo Programs & Demo Browser

The over 300 Demo Programs will give you a jump-start and provide many design patterns for you to learn how to use PySimpleGUI and how to integrate PySimpleGUI with other packages. By far the best way to experience these demos is using the Demo Browser. This tool enables you to search, edit and run the Demo Programs.

To get them installed quickly along with the Demo Browser, use `pip` to install `psgdemos`:

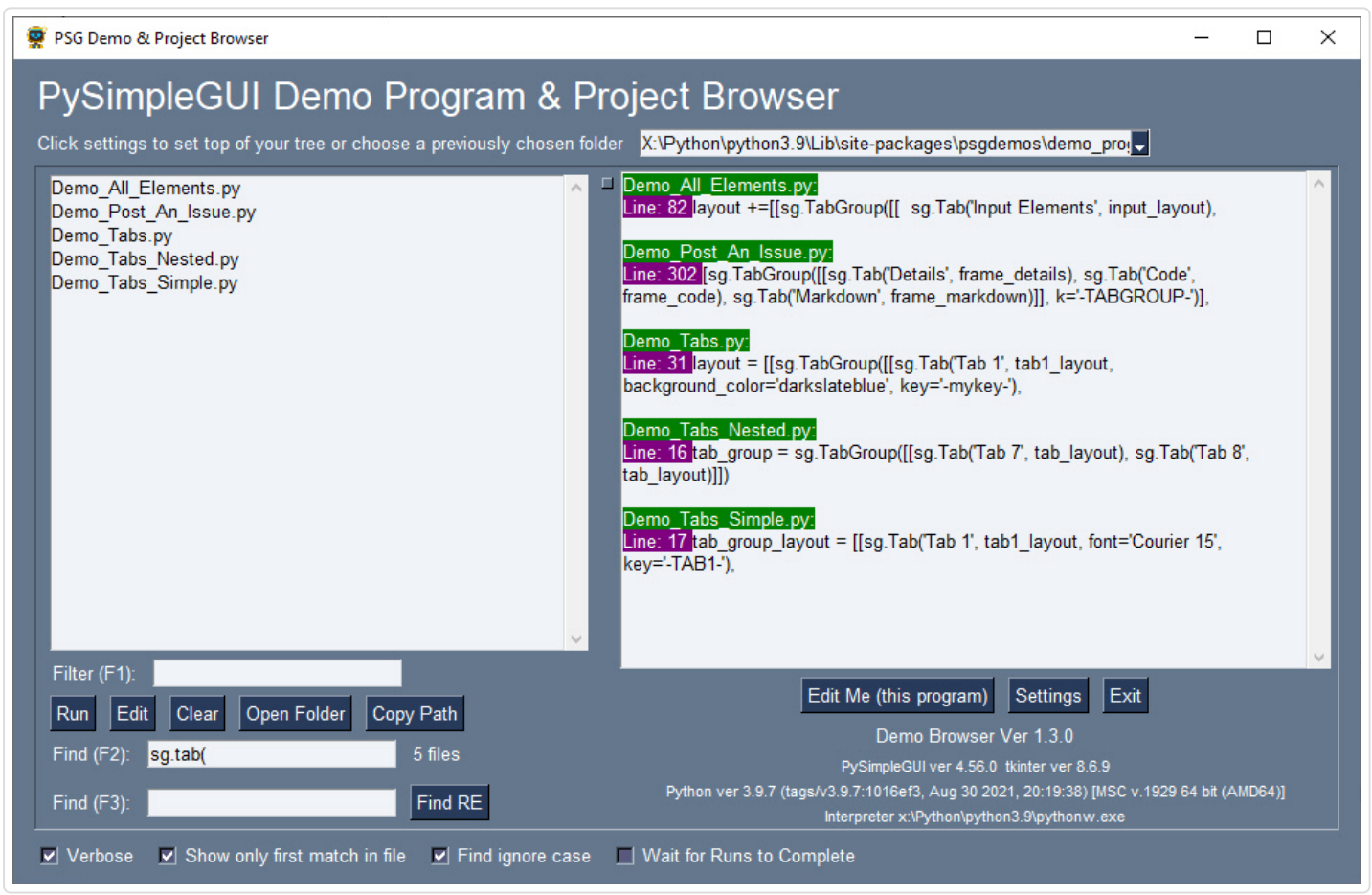
```
python -m pip install psgdemos
```

or if you're in Linux, Mac, etc, that uses `python3` instead of `python` to launch Python:

```
python3 -m pip install psgdemos
```

Once installed, launch the demo browser by typing `psgdemos` from the command line"

```
psgdemos
```

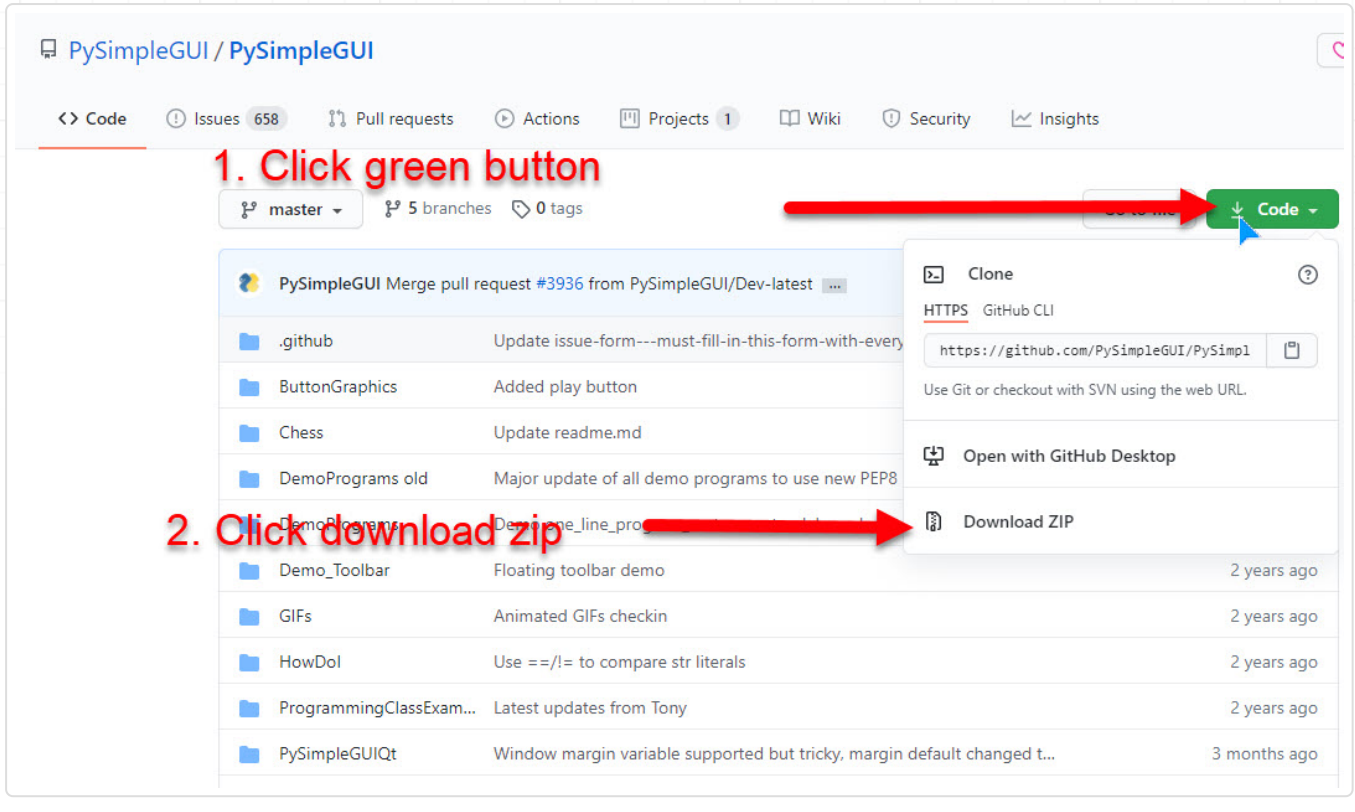


Download the Repo and Demos

If you chose not to pip install the `psgdemos` package, then you can download the PySimpleGUI Repo from GitHub and run the Demo Browser.

Follow these instructions to download the repo and demos:

- Go to http://www.PySimpleGUI.com (the PySimpleGUI GitHub)
- Download the repo as a ZIP file



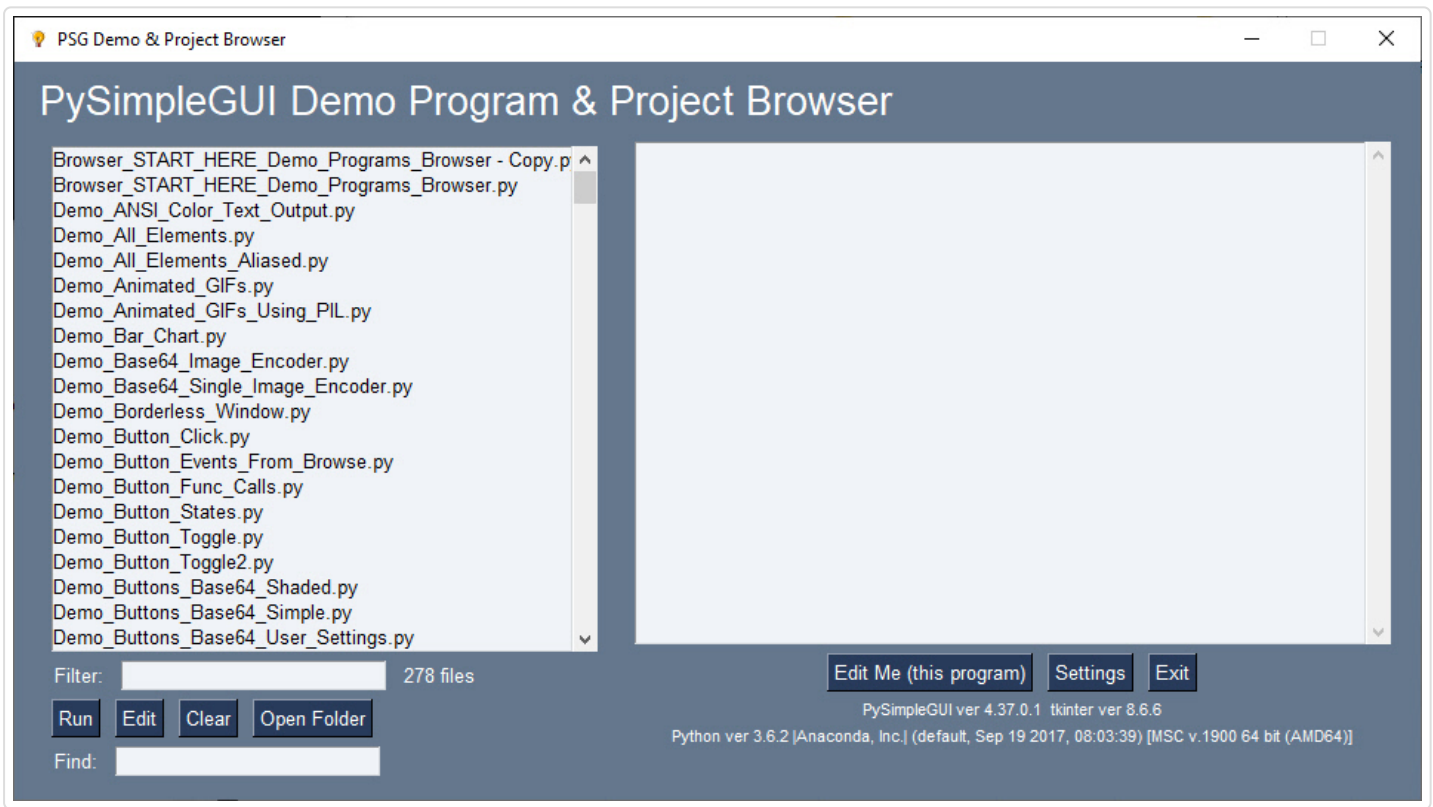
- Unzip the downloaded zip and place the folder `DemoPrograms` somewhere on your local disk drive that you have write access

Setup the Demo Browser

Once you've got your Demo Programs folder set up on your local computer:

- Run the program `Browser_START_HERE_Demo_Programs_Browser.py` that is in the folder you unzipped

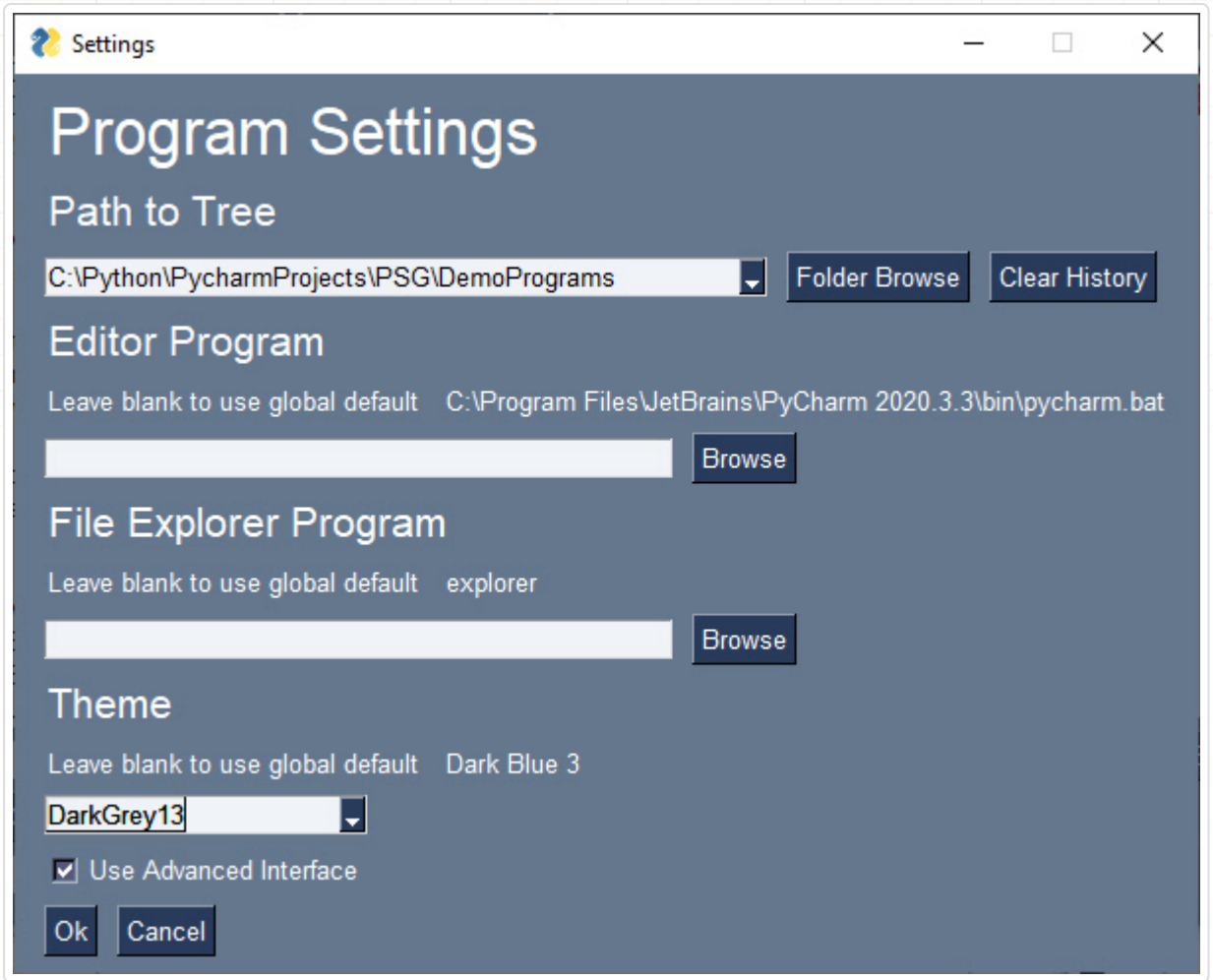
When you first run the Demo Browser, it will probably look something like this:



To change the settings:

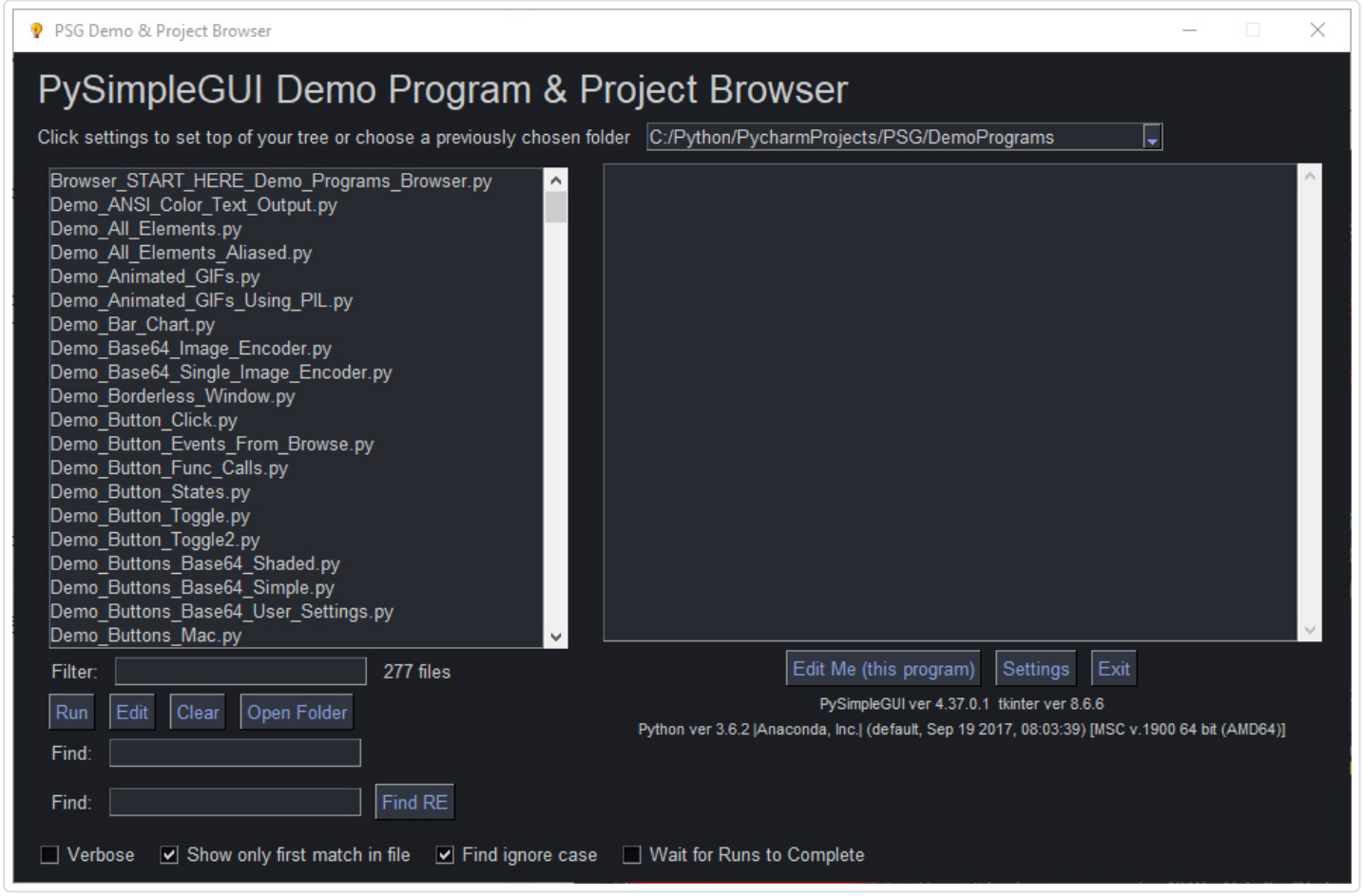
- Click the "Settings" button

Click the "Settings" button. You'll see a window like this one:



I've changed the theme to be "DarkGrey13" and checked "Use Advanced Interface". I know a number of users like "Dark Themes" so let's use a "dark grey 13" theme for this recipe and enable the advanced features.

After clicking OK the window changes to:



More Details on Settings

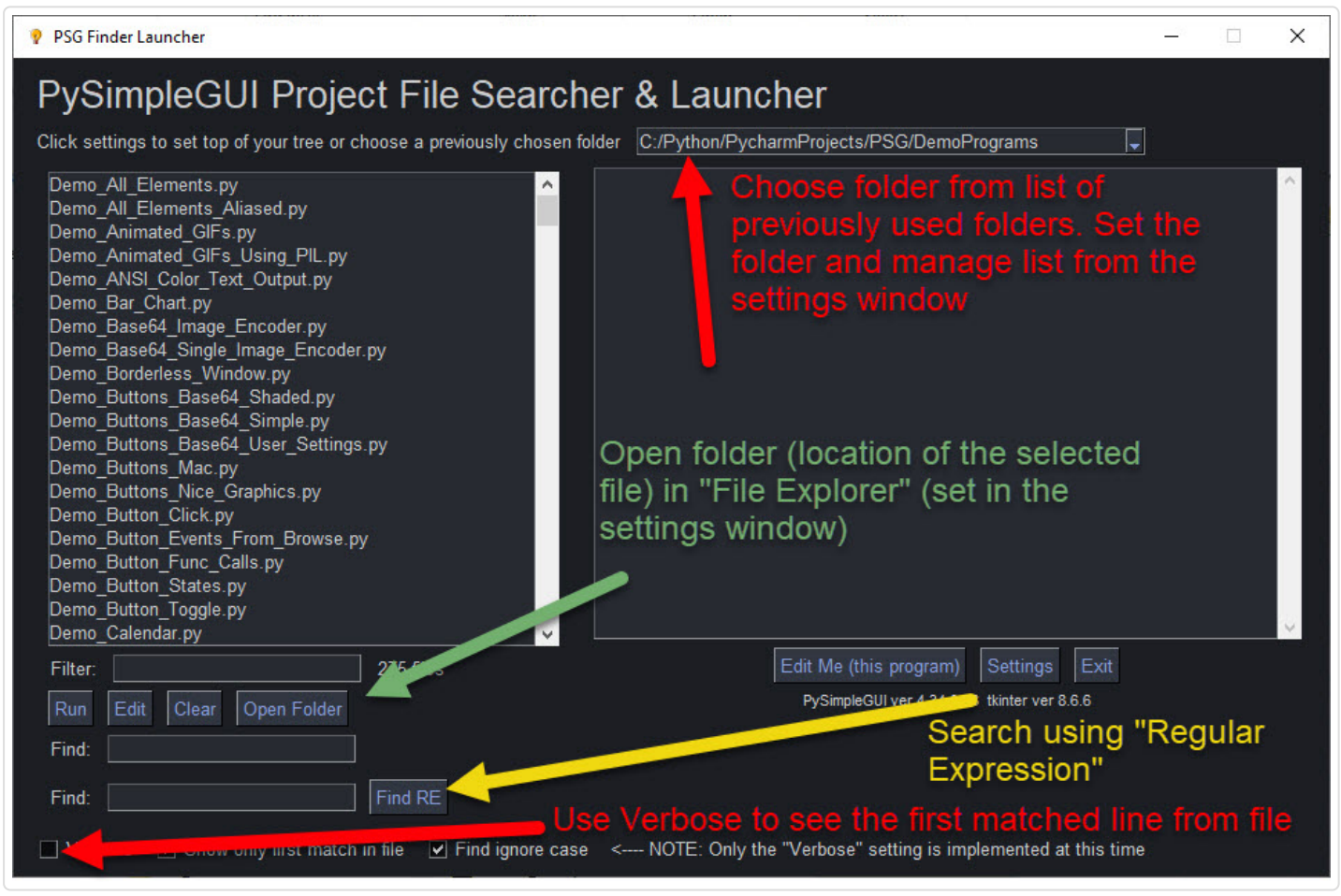
- Fill in the input fields labelled Path to Demos and Editor Program
- Path to Demos - Defaults to the location of the Demo when you run it the first time. You can leave unchanged if this program is going to remain with the other demos. Use an absolute path.
- Editor Program - If you want to be able to use the "Edit" button in this browser to open and edit the demo programs, then fill in the name of the .EXE (for windows) of your editor/IDE. If you have set up an editor in the global settings for PySimpleGUI, then you don't need to set up an editor in the browser demo.
    - PyCharm

        - Windows - it's not an EXE but a batch file. It will look something like this: `C:\Program Files\JetBrains\PyCharm Community Edition 2020.3\bin\pycharm.bat`
        - Linux - it's a bit trickier. It may resemble something like this: `/home/mike/.local/share/JetBrains/Toolbox/apps/PyCharm-C/ch-0/202.8194.15/bin/pycharm.sh`
        - Mac - I dunno yet.... there's an open Issue on GitHub asking for help from Mac user
    - Notepad++
        - Windows - Something along the lines of `C:\Program Files\NotePad++\notepad++.exe`
    - Notepad
        - Windows - `c:\windows\notepad`
        - Linux - `/usr/bin/notepad`
    - Custom
        - Assuming your editor is invoked using "editor filename" when you can use any editor you wish by filling in the "Editor Program" field with a path to the editor executable

NOTE - if you want to be able to use the Edit button to open to a specific line number, then you will need to setup the editor in the PySimpleGUI Global Settings. You can access the PySimpleGUI Global Settings from Test Harness by calling `sg.main()`. There is a button labelled "Global Settings". There is already a lot of duplication of settings happening between this browser and the PySimpleGUI global settings. Once you've got the global settings made, then you don't need to make any chnages to the seettings in the browser program. You can leave them all blank and the settings will come from the PySimpleGUI global settings.

Window Sections

These are some of the important part of the window that are not as obvious as other parts.



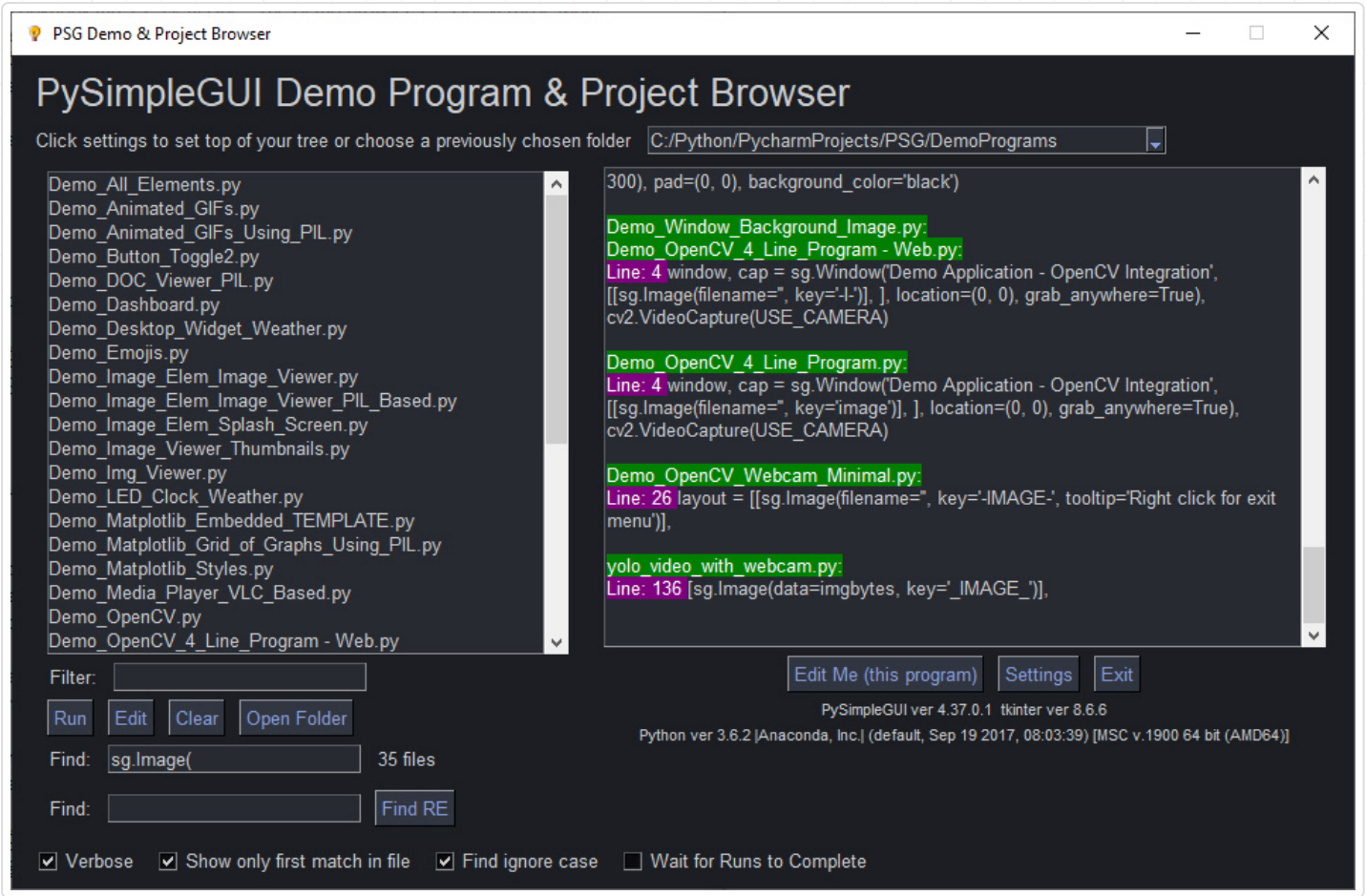
Verbose Mode

Verbose mode will show you the full matching line from the file. This is handy for when you're looking for a specific way something is used.

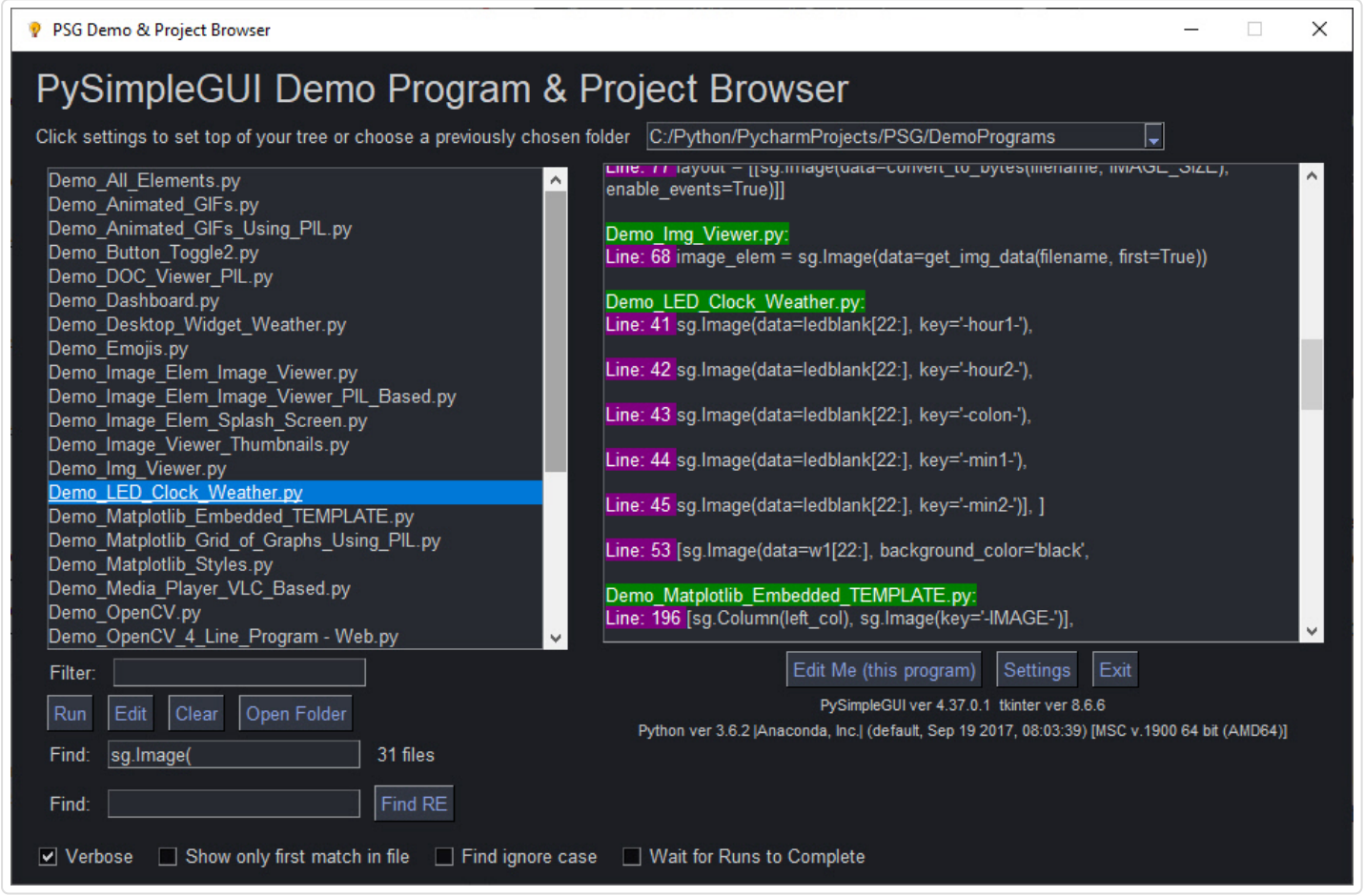
By default the Verbose mode is turned off. This is because search results are updated in realtime as you type characters. If your project tree is large then your verbose output will be very very large as you type the first few characters of your search.

The best way to use the Verbose feature is to perform the search with it turned off. Once you get the list of files from your search, then click verbose to see the more detailed view of the results.

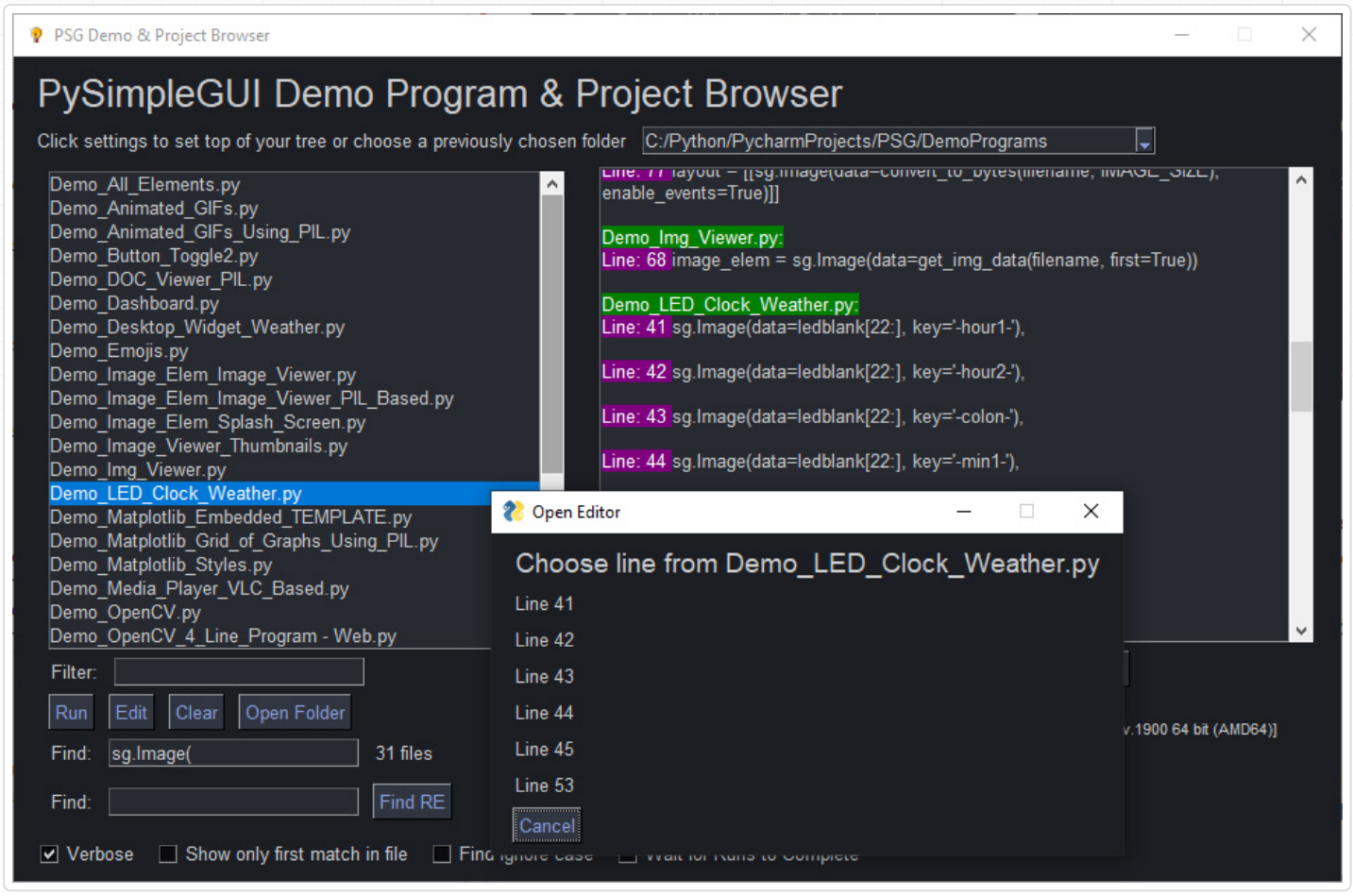
In this example, I've searched for the demos that use the `Graph` element. Searching for "sg.Image(" will return the demos that make this element.



If I then uncheck the "Show onlyu first match in file" then I can see when multiple matches are found in a file.



In verbose mode and all matches are shown, when a file is selected from the list and the "Edit" button is clicked, then another window will show you the list of lines that matched and allows you to click on the line to be taken to. Selecting a line will open your editor to that line.



NOTE - this feature is 99% complete! The lines shwon in the "Open Editor" window don't yet show the code that is contained on each of those lines. Comoing VERY SOON!

"Open Folder" feature

If you have an explorer program specified in the settings or in the PySimpleGUI Global Settings, then choosing a file and clocking the "Open Folder" button will launch the file browser that you've specified and open the folder that the file is contained in.

Recipe - A Simple & Standard Right Click Menu

You'll find that many/most of the PySimpleGUI Demo Programs that are newer have a standard right click menu added to them with these 3 items:

- Edit Me
- Version
- Exit

Adding the Menu

There are several pre-defined right-click menus included with PySimpleGUI. To get the one that includes these 3 options, add this parameter to your `Window` creation call.

```
right_click_menu=sg.MENU_RIGHT_CLICK_EDITME_VER_EXIT
```

If you right click your Window, you'll see a menu that looks something like this:

In the PySimpleGUI code, the constant `MENU_RIGHT_CLICK_EDITME_VER_EXIT` makes this menu definition:

```
MENU_RIGHT_CLICK_EDITME_VER_EXIT = ['', ['Edit Me', 'Version', 'Exit']]
```

Edit Me

When you're developing your code and even after it's done, sometimes you'll see something while the code is running that triggers you to want to make a change. Maybe it's a bug fix, a mispelling (sic), or a new feature idea.

The problem you face now is.... where's the source code?

The Edit Me item launches the editor you've specified in the PySimpleGUI Global Settings and opens the file in that editor.

2 mouse clicks and you're editing your code.

In order for this feature to work, you'll need to add these 2 lines to your event loop:

```
        if event == 'Edit Me':
            sg.execute_editor(__file__)
```

You don't need to modify them. Just add them as they are to your PySimpleGUI programs

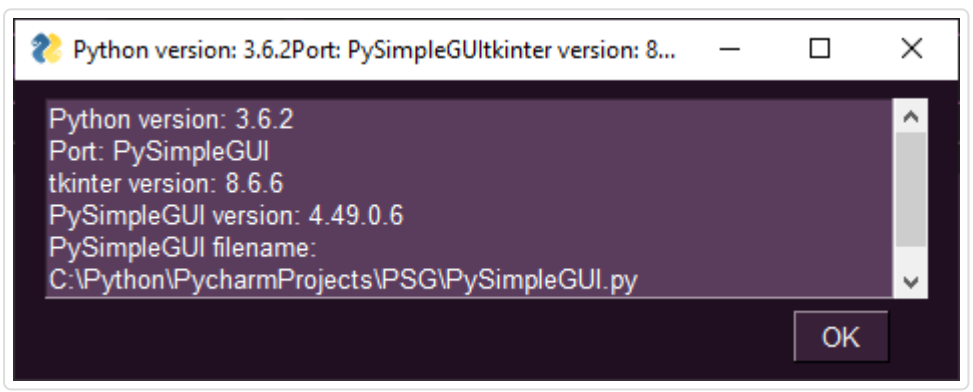
Version

This is another runtime question that is often handy to know without having to look - "What version of PySimpleGUI, Python and tkinter is this running again?"

To add this feature, add these 2 lines to your event loop:

```
        if event == 'Version':
            sg.popup_scrolled(sg.get_versions())
```

You'll see something like this:



Exit

This one is easy. It's particularly good for "Desktop Widgets" that have no titlebar and thus have no "X" to click to exit the program. If your event loop already has a change for the event "Exit" then you don't need to do anything to get this feature.

Recipe - No Console Launching

Windows 10

You've just coded up your nice GUI, ready for that "Windows experience".... you double click your .py file in Windows explorer and up comes 2 windows, a console and your GUI Window.

If you're ready for a more Windows-like experience for you and your users, then these steps should get you there.

There are a couple of simple tricks needed.

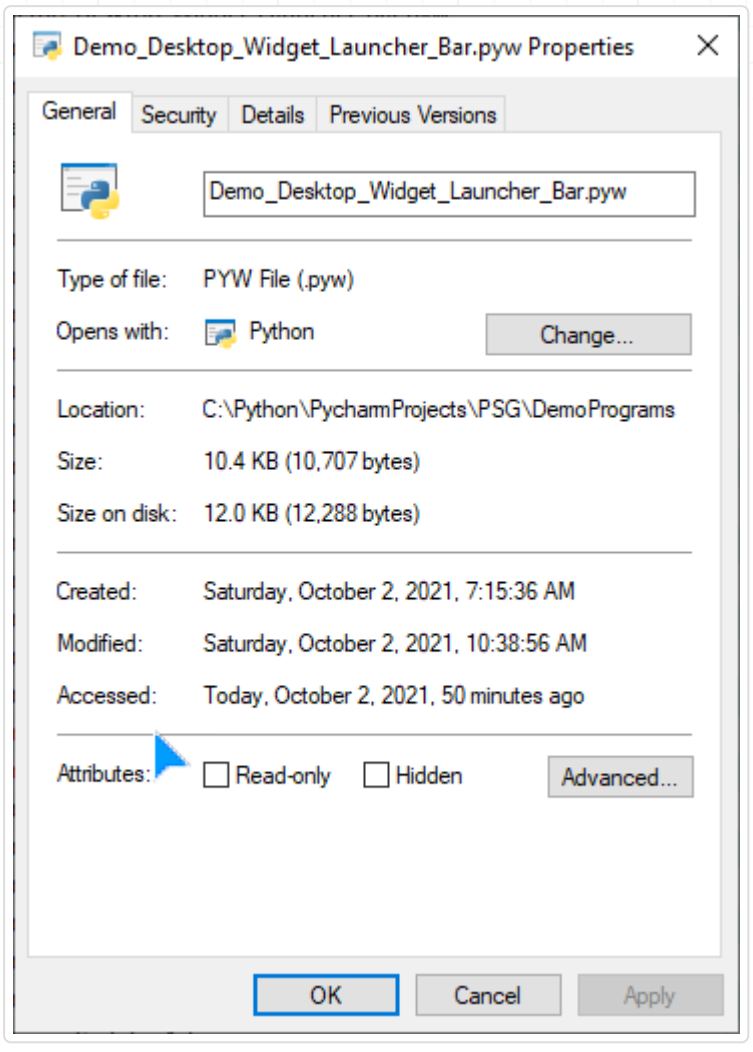
Rename Your `.py` to `.pyw`

1. Rename your .py file to .pyw
2. Launch it using pythonw.exe instead of python.exe

When you right click on your .pyw file, you'll want to associate it with `pythonw.exe` if that's not already been done by Python for you when it was installed.

Right click your .pyw file and choose "Properties"

You'll see a window like this which will tell you what will open the file if you double click it. You can click the "Change..." button to change what opens the file.



Make A Shortcut

You've learned how to use explorer to double-click and launch your GUI window without a console. Let's go 1 step further.

If you were to simply drag your .pyw file to your taskbar to "pin" it for quick launching, unfortunately that doesn't work. It'll try and launch Python instead.

What we're going to do is make an icon/shortcut to your python program that you can place anywhere AND you can also pin it to your taskbar.

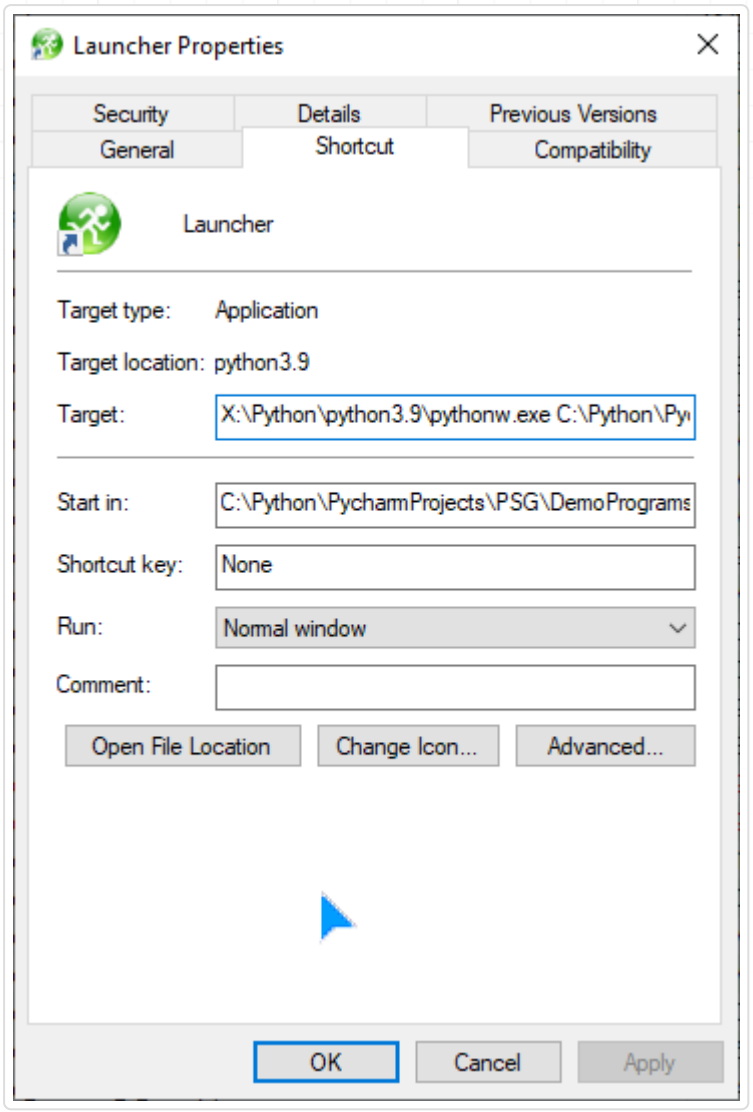
To make a shortcut, right click your .pyw file and choose "Create Shortcut". It's near the end of the list of right click options

My program is called `Demo_Desktop_Widget_Launcher_Bar.pyw`. After I created the shortcut, a file named `Demo_Desktop_Widget_Launcher_Bar.pyw - Shortcut` was created in the same folder.

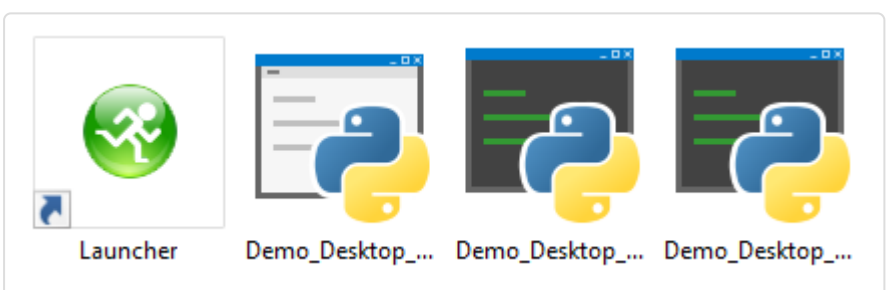
You should now do these 2 operations on that shortcut 1. Rename it something that looks like any other program 2. Choose an icon for your shortcut. On windows, this needs to be a .ICO file 3. Set a specific program that opens it (your pythonw.exe file)

I named my shortcut "Launcher".

Because I wanted the 3.9 version of Python to open this specific program, I added pythonw.exe part to the target field in the properties. After the pythonw.exe add your full path to your .pyw file. If there are any spaces, put "" around it. It should look something like this:



In my folder with my program, I now see the shortcut with the icon I chose earlier



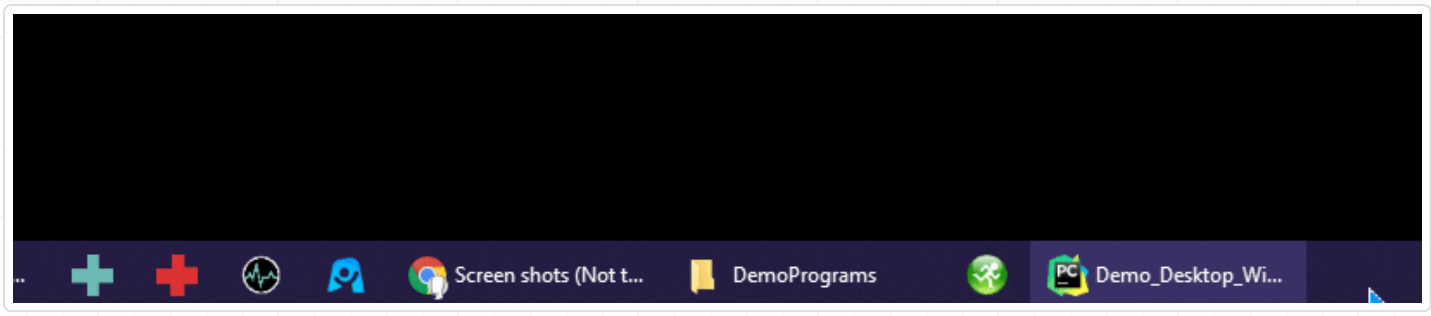
Double-clicking this icon will start your .pyw file without a console. It will look "like a real windows program"

Pin Your Shortcut

If you want to be able to quickly launch your program, you can "pin" your shortcut to your taskbar. To do this, you can drag and drop the icon onto the taskbar. Or, you can right click the icon and choose "pin to taskbar".

After pinning it, I see this as my taskbar. The launcher icon is there ready to be clicked. When I click it, the program runs without any console windows.

This particular .pyw file is the Demo Launcher Bar. Clicking the icon launches this launcher.



Themes - Window "Beautification"

"Beautiful windows" don't just happen, but coloring your window can be accomplished with 1 line of code.

One complaint about tkinter that is often heard is how "ugly" it looks. You can do something about that by using PySimpleGUI themes.

```
sg.theme('Dark Green 5')
```

A call to `theme` will set the colors to be used when creating windows. It sets text color, background color, input field colors, button color,.... 13 different settings are changed.

The default theme is "Dark Blue 3"

Look and Feel Theme Explosion

There are currently 140 themes to choose from (in April 2020, maybe more by the time you read this)

To see the above preview for your version of PySimpleGUI, make this call to generate a preview of all available themes:

```
sg.theme_previewer()
```

Even windows that are created for you, such as popups, will use the color settings you specify. And, you can change them at any point, even mid-way through defining a window layout.

This one line of code helps, but it's not the only thing that is going to make your window attractive.

Theme Name Format

You can look at the table of available themes to get the name of a theme you want to try, or you can "guess" at one using this formula:

```
<"Dark" | "Light"> <Color> [#]
```

Where Color is one of these:

```
Black, Blue, Green, Teal, Brown, Yellow, Gray, Purple
```

The # is optional and is used when there is more than 1 choice for a color. For example, for "Dark Blue" there are 12 different themes (Dark Blue, and Dark Blue 1-11). These colors specify the rough color of the background. These can vary wildly so you'll have to try them out to see what you like the best.

Recipe - Built-in Theme Viewer

If you want to see a window on your system like the above theme preview screenshot, then make this call and you'll see the same window:

```
import PySimpleGUI as sg

sg.preview_all_look_and_feel_themes()
```

Specifying and Getting Theme Names

In addition to getting all of these new themes, the format of the string used to specify them got "fuzzy". You no longer have to specify the ***exact*** string shown in the preview. Now you can add spaces, change the case, even move words around and you'll still get the correct theme.

For example the theme `"DarkBrown2"` can be specified also as `"Dark Brown 2"`.

If you can't remember the names and get it wrong, you'll get a text list of the available choices printed on your console.

You can also get the list of theme names by calling `theme_list`

```
import PySimpleGUI as sg

theme_name_list = sg.theme_list()
```

If you guess incorrectly, then you'll be treated to a random theme instead of some hard coded default. You were calling theme to get more color, in theory, so instead of giving you a gray window, you'll get a randomly chosen theme (and you'll get the name of this theme printed on the console). It's a great way to discover new color combinations via a mistake.

Recipe - Post your screen-shots (PLEASE!)

This is an odd recipe, but it's an important one and has nothing to do with your coding PySimpleGUI code. Instead is has to do with modifying youre readme.md file on your GitHub so you can share with the world your creation. Doon't be shy. We all stated with "hello world" and your first GUI is likely to be primitive, but it's very important you post it any way.

In case you've not noticed, you, the now fancy Python GUI programmer that you are, are a rare person in the Python world. The VAST majority of Python projects posted on GitHub do not contain a GUI. This GUI thing is kinda new and novel for Python pbeginning rogrammers.

People / visitors love pictures

They don't have to be what you consider to be "pretty pictures" or of a "compex GUI". GUIs from beginners should be shown as proudly developed creations you've completed or are in the process of completion.

Your GitHub visitors may never have made a GUI and need to see a beginner GUI just as much as they need to see more complex GUIs. It gives them a target. It shows them someone they may be able to achieve.

The GitHub Issue Technique

This is one of the easiest / laziest / quickest ways of adding a screenshot to your Readme.md and this post on your project's main page.

Here'show you do it:

1. Open a "Screenshots" Issue somehwere in GitHub. It sdoesn't matter which project you open it under.
2. Copy and paste your image into the Issue's comment section. OR Drag and drop your image info the comment section. OR click the upload diaload box by clickin at the bottom on the words "Attach files by dragging & dropping, selecting or pasting them.
3. A line of code will be inserted when you add a the image to your GitHub Issue's comment. The line of code will resemble this:

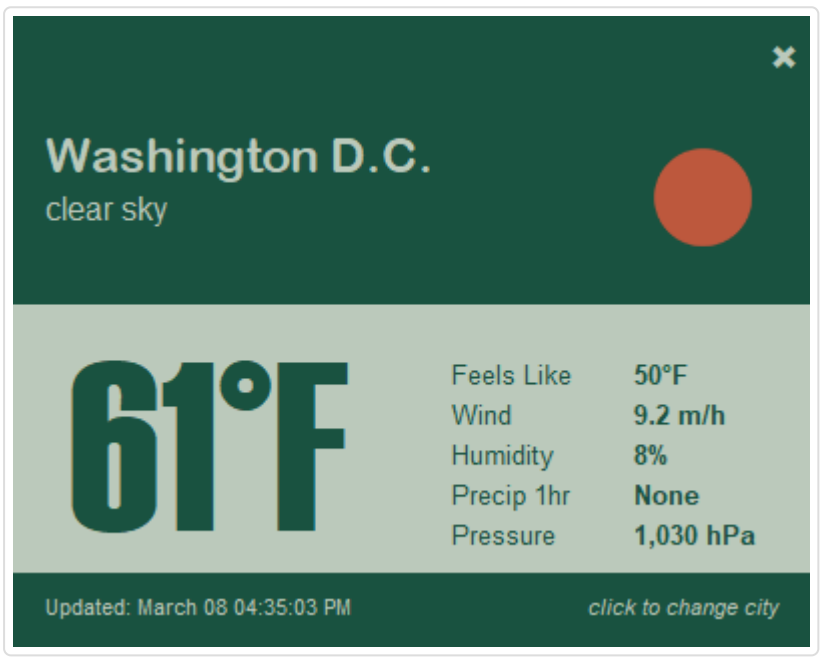
```
![image](https://user-images.githubusercontent.com/46163555/76170712-d6633c00-615a-11ea-866a-11d0b
```

1. Copy the line of code that is created in the comment. You can see this line when in the "Write" mode for the Issue.. If you want to see how it'll look, switch to the preview tab.
2. Paste the line of code from the Issue Comment into your readme.md file located in your mtop-level FirHub folder

That's it.

Note, if you simply copy the link to the image that's created, in your readme.md file you will see only the link. The image will not be embedded into the page, only the link will be shown The thing you paste into your readme needs to have this format, theat starts with `![filename]`.

Pasting the above line directly into this Cookbook resulted in this Weahter Widget posted::



The Image Hosted Elsehwere Technique

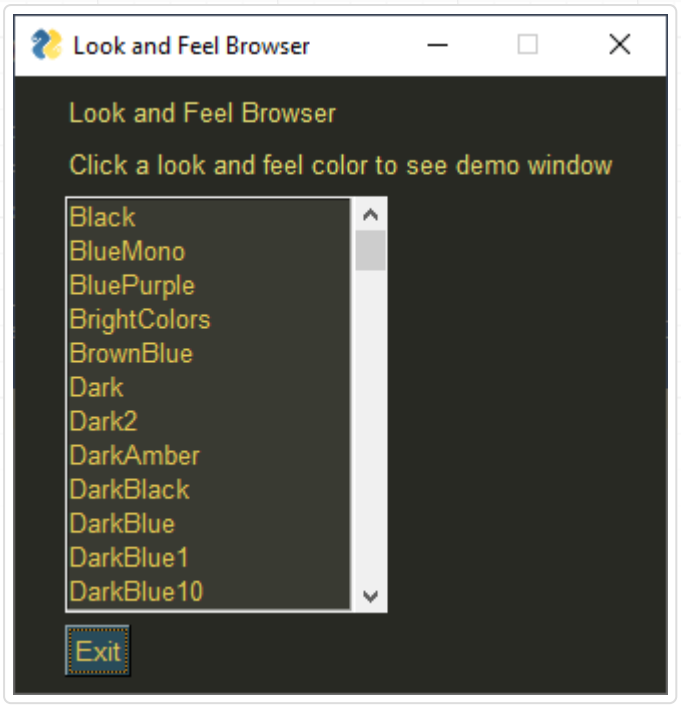
The same technique is used as above, except in the line of code, you'll insert the URL of your image where every that may be inside the `( )`

```
![image](http://YourLinkToYourImage.jpg)
```

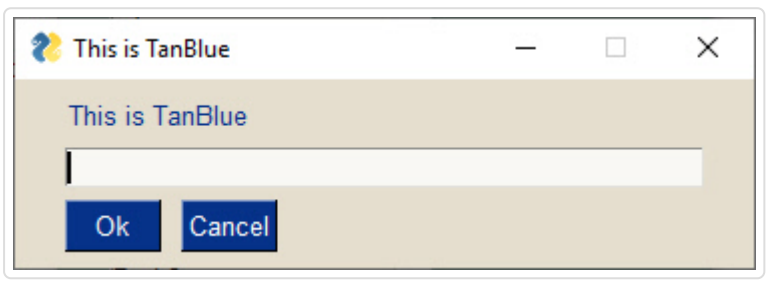
Recipe - Theme Browser

This Recipe is a "Theme Browser" that enables you to see the different color schemes.

You're first shown this window that lists all of the available "Theme" settings. This window was created after the Theme were set to "Dark Brown".



If you click on an item in the list, you will immediately get a popup window that is created using the new Theme.




```
import PySimpleGUI as sg

"""
    Allows you to "browse" through the Theme settings.  Click on one and you'll see a
    Popup window using the color scheme you chose.  It's a simple little program that also demonst
    how snappy a GUI can feel if you enable an element's events rather than waiting on a button cl
    In this program, as soon as a listbox entry is clicked, the read returns.
"""

sg.theme('Dark Brown')


layout = [[sg.Text('Theme Browser')],
          [sg.Text('Click a Theme color to see demo window')],
          [sg.Listbox(values=sg.theme_list(), size=(20, 12), key='-LIST-', enable_events=True)],
          [sg.Button('Exit')]]

window = sg.Window('Theme Browser', layout)

while True:  # Event Loop
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    sg.theme(values['-LIST-'][0])
    sg.popup_get_text('This is {}'.format(values['-LIST-'][0]))

window.close()
```

Making Changes to Themes & Adding Your Own Themes

Modifying and creating your own theme is not difficult, but tricky so start with something and modify it carefully.

The tkinter port has the theme_add_new function that will add a new dictionary entry into the table with the name you provide. It takes 2 parameters - the theme name and the dictionary entry.

The manual way to add a dictionary entry is as follows....

The Theme definitions are stored in a dictionary. The underlying dictionary can be directly accessed via the variable `LOOK_AND_FEEL_TABLE`.

A single entry in this dictionary has this format (copy this code):

```
'LightGreen3': {'BACKGROUND': '#A8C1B4',
                'TEXT': 'black',
                'INPUT': '#DDE0DE',
                'SCROLL': '#E3E3E3',
                'TEXT_INPUT': 'black',
                'BUTTON': ('white', '#6D9F85'),
                'PROGRESS': DEFAULT_PROGRESS_BAR_COLOR,
                'BORDER': 1,
                'SLIDER_DEPTH': 0,
                'PROGRESS_DEPTH': 0}
```

As you can see, a single entry in the Look and Feel dictionary is itself a dictionary.

Recipe - Modifying an existing Theme

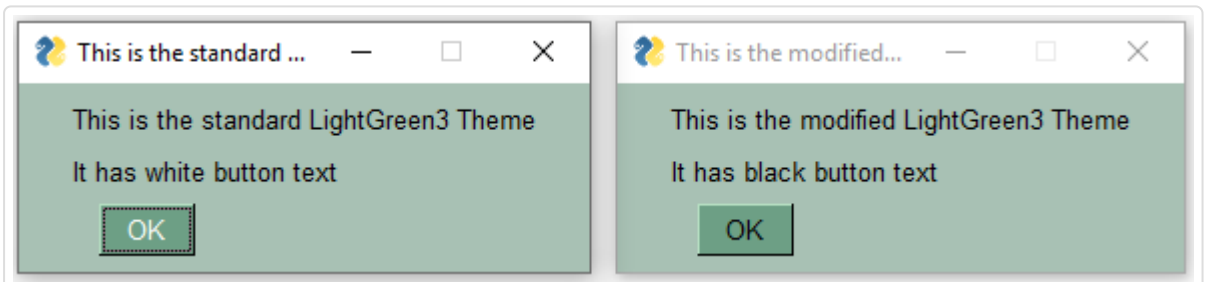
Let's say you like the `LightGreeen3` Theme, except you would like for the buttons to have black text instead of white. You can change this by modifying the theme at runtime.

Normal use of `theme` calls is to retrieve a theme's setting such as the background color. The functions used to retrieve a theme setting can also be used to modify the setting by passing in the new setting as a parameter.

Calling `theme_background_color()` returns the background color currently in use. Passing in the color `'blue'` as the parameter, `theme_background_color('blue')`, will change the background color for future windows you create to blue.

```
import PySimpleGUI as sg

sg.theme('LightGreen3')
sg.popup_no_wait('This is the standard LightGreen3 Theme', 'It has white button text')

# Modify the theme
sg.theme_button(('black', '#6D9F85'))

sg.popup('This is the modified LightGreen3 Theme', 'It has black button text')
```

Produces these 2 windows

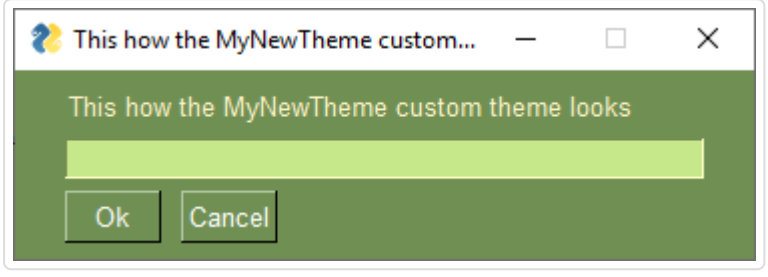


Recipe - Adding Your Own Color Theme

The great thing about these themes is that you set it onces and all future Elements will use the new settings. If you're adding the same colors in your element definitions over and over then perhaps making your own theme is in order.

Let's say that you need to match a logo's green color and you've come up with matching other colors to go with it. To add the new theme to the standard themes this code will do it:

```
import PySimpleGUI as sg

# Add your new theme colors and settings
my_new_theme = {'BACKGROUND': '#709053',
                'TEXT': '#fff4c9',
                'INPUT': '#c7e78b',
                'TEXT_INPUT': '#000000',
                'SCROLL': '#c7e78b',
                'BUTTON': ('white', '#709053'),
                'PROGRESS': ('#01826B', '#D0D0D0'),
                'BORDER': 1,
                'SLIDER_DEPTH': 0,
                'PROGRESS_DEPTH': 0}

# Add your dictionary to the PySimpleGUI themes
sg.theme_add_new('MyNewTheme', my_new_theme)

# Switch your theme to use the newly added one. You can add spaces to make it more readable
sg.theme('My New Theme')

# Call a popup to show what the theme looks like
sg.popup_get_text('This how the MyNewTheme custom theme looks')
```



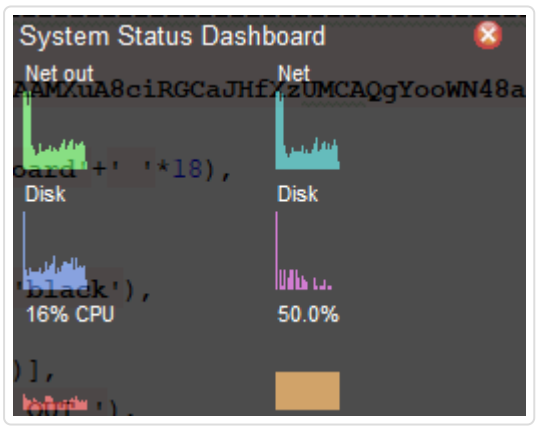
More Ways to "Dress Up Your Windows"

In addition to color there are a several of other ways to potentially make your window more attractive. A few easy ones include:

- Remove the titlebar
- Make your window semi-transparent (change opacity)
- Replace normal buttons with graphics

You can use a combination of these 3 settings to create windows that look like Rainmeter style desktop-widgets.

This window demonstrates these settings. As you can see, there is text showing through the background of the window. This is because the "Alpha Channel" was set to a semi-transparent setting. There is no titlebar going across the window and there is a little red X in the upper corner, resumably to close the window.



Recipe Removing the Titlebar & Making Semi-Transparent

Both of these can be set when you create your window. These 2 parameters are all you need - `no_titlebar` and `alpha_channel`.

When creating a window without a titlebar you create a problem where the user is unable to move your window as they have no titlebar to grab and drag. Another parameter to the window creation will fix this problem - `grab_anywhere`. When `True`, this parameter allows the user to move the window by clicking anywhere within the window and dragging it, just as if they clicked the titlebar. Some PySimpleGUI ports allow you to click on input fields and drag, others require you to grab a spot on the background of the window. Note - you do not have to remove the titlebar in order to use `grab_anywhere`

To make your window semi-transpaerent (change the opacity) use ghe `alhpa_channel` parameter when you create the window. The setting is a float with valid values from 0 to 1.

To create a window like the one above, your window creation call would look something like:

```
window = sg.Window('PSG System Dashboard', layout, no_titlebar=True, alpha_channel=.5, grab_anywhe
```

Recipe - Replacing a Button with a Graphic

In PySimpleGUI you can use PNG and GIF image files as buttons. You can also encode those files into Base64 strings and put them directly into your code.

It's a 4 step process to make a button using a graphic

1. Find your PNG or GIF graphic
2. Convert your graphic into a Base64 byte string
3. Add Base64 string to your code as a variable
4. Specify the Base64 string as the image to use when creating your button

Step 1 - Find your graphic

There are a LOT of places for you to find your graphics. This page lists a number of ways to search for what you need. Bing also has a great image search tool that you can filter your results on to get a list of PNG files (choose "Transparent" using their "filter" on the page.)

Here's the search results for "red x icon" using Bing with a filter.

I chose this one from the list: http://icons.iconarchive.com/icons/iconarchive/red-orb-alphabet/256/Letter-X-icon.png

You can download your image or get a copy of the link to it.

Step 2 - Convert to Base64

One of the demo programs provided on the PySimpleGUI GitHub is called "Demo_Base64_Image_Encoder.py". This program will convert all of the images in a folder and write the encoded data to a file named `output.py`.

Another demo program, "Demo_Base64_Single_Image_Encoder.py" will convert the input file to a base64 string and place the string onto the clipboard. Paste the result into your code and assign it to a variable.

Both are in the Demos folder (Demos.PySimpleGUI.org)

Step 3 - Make Base64 String Variable

Select all of the data in the Base64 box and paste into your code by making a variable that is equal to a byte-string.

```
red_x_base64 = b''
```

Paste the long data you got from the webpage inside the quotes after the `b`.

You can also copy and paste the byte string from the `output.py` file if you used the demo program or paste the string created using the single file encoder demo program.

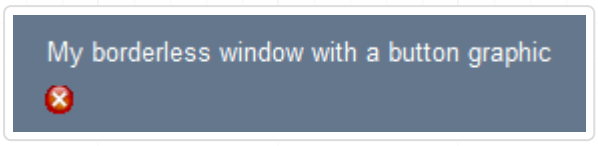
Step 4 - Use Base64 Variable to Make Your Button

This is the Button Element that is added to the layout to create the Red X Button graphic.

You need to set the background color for your button to be the same as the background the button is being placed on if you want it to appear invisible.

```
sg.Button('', image_data=red_x_base64,
          button_color=(sg.theme_background_color(),sg.theme_background_color()),
          border_width=0, key='Exit')
```

This is the window the code below creates using a button graphic.



You can [run similar code online on Trinket]

```
import PySimpleGUI as sg
# Note that the base64 string is quite long.  You can get the code from Trinket that includes the
red_x_base64 = b'paste the base64 encoded string here'
red_x_base64 = sg.red_x     # Using this built-in little red X for this demo

layout = [  [sg.Text('My borderless window with a button graphic')],
            [sg.Button('', image_data=red_x_base64,
            button_color=(sg.theme_background_color(),sg.theme_background_color()),border_width=0,

window = sg.Window('Window Title', layout, no_titlebar=True)

while True:             # Event Loop
    event, values = window.read()
    print(event, values)
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
window.close()
```

When working with PNG/GIF files as button images the background you choose for the button matters. It should match the background of whatever it is being placed upon. If you are using the standard "themes" interfaces to build your windows, then the color of the background can be found by calling `theme_background_color()`. Buttons have 2 colors so be sure and pass in TWO color values when specifying buttons (text color, background color).

Recipe - 1 Shot Window - Simple Data Entry - Return Values - Auto Numbered

Remember how keys are **key** to understanding PySimpleGUI elements? Well, they are, so now you know.

If you do not specify a key and the element is an input element, a key will be provided for you in the form of an integer, starting numbering with zero. If you don't specify any keys, it will appear as if the values returned to you are being returned as a list because the keys are sequential ints.

This example has no keys specified. The 3 input fields will have keys 0, 1, 2. Your first input element will be accessed as `values[0]`, just like a list would look.

```
import PySimpleGUI as sg

sg.theme('Topanga')      # Add some color to the window

# Very basic window.  Return values using auto numbered keys

layout = [
    [sg.Text('Please enter your Name, Address, Phone')],
    [sg.Text('Name', size=(15, 1)), sg.InputText()],
    [sg.Text('Address', size=(15, 1)), sg.InputText()],
    [sg.Text('Phone', size=(15, 1)), sg.InputText()],
    [sg.Submit(), sg.Cancel()]
]

window = sg.Window('Simple data entry window', layout)
event, values = window.read()
window.close()
print(event, values[0], values[1], values[2])    # the input data looks like a simple list when au
```

Recipe - Add GUI to Front-End of Script

Quickly add a GUI allowing the user to browse for a filename if a filename is not supplied on the command line using this simple GUI. It's the best of both worlds. If you want command line, you can use it. If you don't specify, then the GUI will fire up.

```
import PySimpleGUI as sg
import sys

if len(sys.argv) == 1:
    event, values = sg.Window('My Script',
                    [[sg.Text('Document to open')],
                    [sg.In(), sg.FileBrowse()],
                    [sg.Open(), sg.Cancel()]]).read(close=True)
    fname = values[0]
else:
    fname = sg.argv[1]

if not fname:
    sg.popup("Cancel", "No filename supplied")
    raise SystemExit("Cancelling: no filename supplied")
else:
    sg.popup('The filename you chose was', fname)
```

If you really want to compress your 1-line of GUI code, you can directly access just the entered data by using this single-line-of-code solution. Dunno if it's the safest way to go, but it's certainly the most compact. Single line GUIs are fun when you can get away with them.

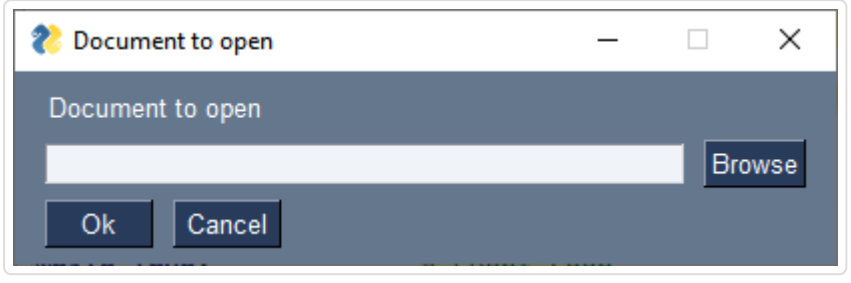
```
import PySimpleGUI as sg
import sys

if len(sys.argv) == 1:
    fname = sg.Window('My Script',
                    [[sg.Text('Document to open')],
                    [sg.In(), sg.FileBrowse()],
                    [sg.Open(), sg.Cancel()]]).read(close=True)[1][0]
else:
    fname = sys.argv[1]

if not fname:
    sg.popup("Cancel", "No filename supplied")
    raise SystemExit("Cancelling: no filename supplied")
else:
    sg.popup('The filename you chose was', fname)
```

Recipe - The `popup_get_file` Version of Add GUI to Front-End of Script

Why recreate the wheel? There's a `Popup` function that will get a Filename for you. This is a single-line GUI:

```
fname = sg.popup_get_file('Document to open')
```

Shows this window and returns the results from the user interaction with it.



The entire Popup based solution for this get filename example is:

```
import PySimpleGUI as sg
import sys

if len(sys.argv) == 1:
    fname = sg.popup_get_file('Document to open')
else:
    fname = sys.argv[1]

if not fname:
    sg.popup("Cancel", "No filename supplied")
    raise SystemExit("Cancelling: no filename supplied")
else:
    sg.popup('The filename you chose was', fname)
```

How about a GUI ***and*** traditional CLI argument in 1 line of code?

```
import PySimpleGUI as sg
import sys

fname = sys.argv[1] if len(sys.argv) > 1 else sg.popup_get_file('Document to open')

if not fname:
    sg.popup("Cancel", "No filename supplied")
    raise SystemExit("Cancelling: no filename supplied")
else:
    sg.popup('The filename you chose was', fname)
```

Recipe - Function and Aliases

This is related to the topic of "User Defined Elements" if you care to go look it up.

If you're using PyCharm, this technique works particuarly well because the DocStrings continue to work even after you have created aliases.

Aliases are used a LOT in PySimpleGUI. You'll find that nearly all of the Elements have multiple names that can be used for them. Text Elements can be specified as `Text`, `Txt`, and `T`. This allows you to write really compact code.

You can make your own aliases too. The advantage of you making your own is that they will be in your own name space and thus will not have the typical `sg.` in front of them.

Let's use the `cprint` function as an example.

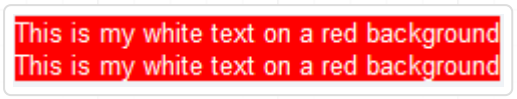
Normally you'll call this function like this:

```
sg.cprint('This is my white text on a red background', colors='white on red')
```

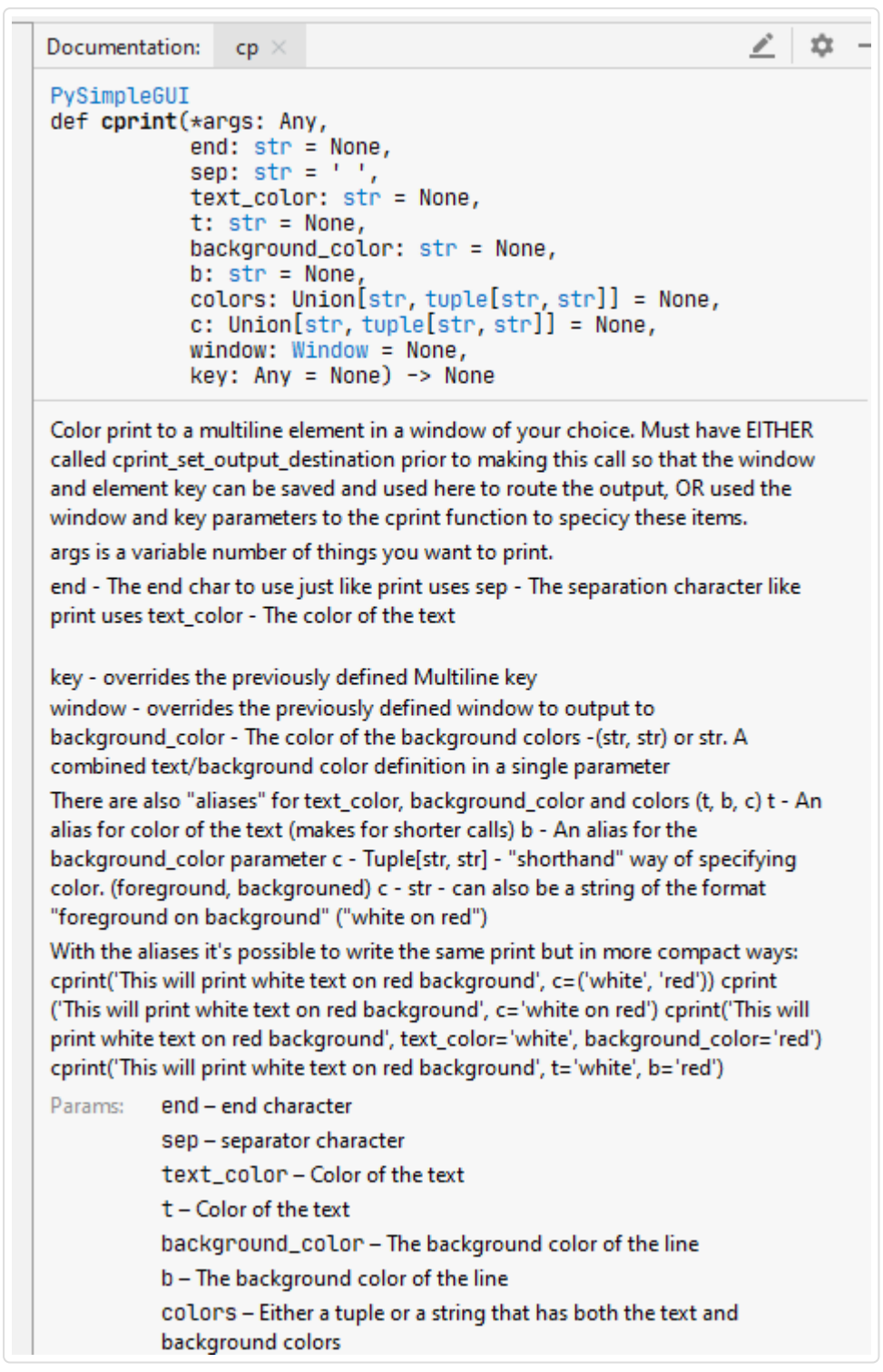
If you have a lot of these in your program, it won't get too long until you're tired of typing `sg.cprint`, so, why not make it super easy on yourself and type `cp` instead. Here's all you have to do.

```
cp = sg.cprint

cp('This is my white text on a red background', colors='white on red')
```

Running these 2 calls produced these 2 lines of text in a Multiline element



IF you're using PyCharm and press Control+Q with your cursor over the `cp`, you'll see the documentation brought up for the `cprint` call:



Feel free to experiment. Even renaming elements will save you the hassle of typing in the `sg.` portion. Then again, so will importing the invdividual elements.

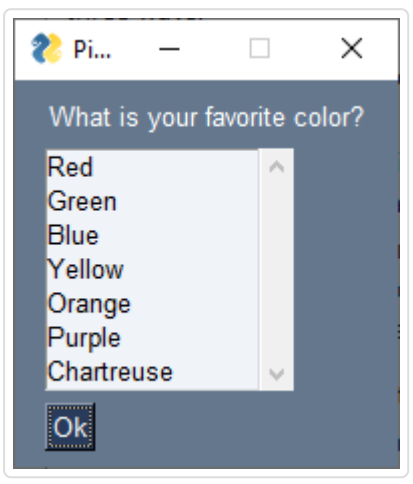
```
# this import will allow you to type just "Text" to use a Text Element
from PySimpleGUI import Text

layout = [[Text('Simpler looking layout')]]
```

Recipe - Highly Responsive Inputs

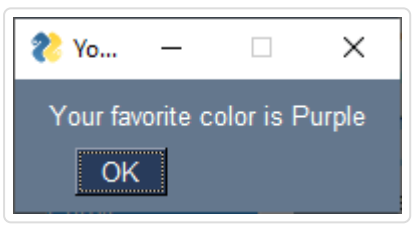
Sometimes it's desireable to begin processing input information when a user makes a selection rather than requiring the user to click an OK button.

Let's say you've got a listbox of entries and a user can select an item from it.

```
import PySimpleGUI as sg

choices = ('Red', 'Green', 'Blue', 'Yellow', 'Orange', 'Purple', 'Chartreuse')

layout = [  [sg.Text('What is your favorite color?')],
            [sg.Listbox(choices, size=(15, len(choices)), key='-COLOR-')],
            [sg.Button('Ok')]  ]

window = sg.Window('Pick a color', layout)

while True:                 # the event loop
    event, values = window.read()
    if event == sg.WIN_CLOSED:
        break
    if event == 'Ok':
        if values['-COLOR-']:    # if something is highlighted in the list
            sg.popup(f"Your favorite color is {values['-COLOR-'][0]}")
window.close()
```



When you choose a color and click OK, a popup like this one is shown:

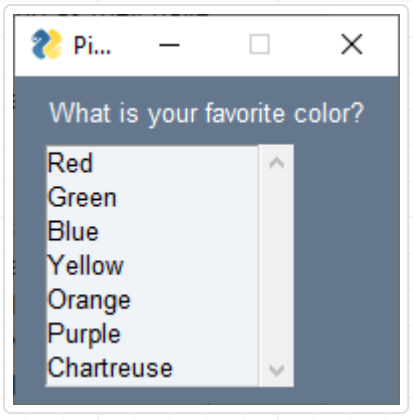


Use `enable_events` to instantly get events

That was simple enough. But maybe you're impatient and don't want to have to cick "Ok". Maybe you don't want an OK button at all. If that's you, then you'll like the `enable_events` parameter that is available for nearly all elements. Setting `enable_events` means that like button presses, when that element is interacted with (e.g. clicked on, a character entered into) then an event is immediately generated causing your `window.read()` call to return.

If the previous example were changed such that the OK button is removed and the `enable_events` parameter is added, then the code and window appear like this:

```
import PySimpleGUI as sg

choices = ('Red', 'Green', 'Blue', 'Yellow', 'Orange', 'Purple', 'Chartreuse')

layout = [  [sg.Text('What is your favorite color?')],
            [sg.Listbox(choices, size=(15, len(choices)), key='-COLOR-', enable_events=True)] ]

window = sg.Window('Pick a color', layout)

while True:                 # the event loop
    event, values = window.read()
    if event == sg.WIN_CLOSED:
        break
    if values['-COLOR-']:    # if something is highlighted in the list
        sg.popup(f"Your favorite color is {values['-COLOR-'][0]}")
window.close()
```



You won't need to click the OK button anymore because as soon as you make the selection in the listbox the `read()` call returns and the popup will be displayed as before.

This second example code could be used with the OK button. It doesn't matter what event caused the `window.read()` to return. The important thing is whether or not a valid selection was made. The check for `event == 'Ok'` is actually not needed.

Recipe - Input Validation

Sometimes you want to restrict what a use can input into a field. Maybe you have a zipcode field and want to make sure only numbers are entered and it's no longer than 5 digits.

Perhaps you need a floating point number and only want to allow `0` - `9`, `.`, and `-`. One way restrict the user's input to only those characters is to get an event any time the user inputs a character and if the character isn't a valid one, remove it.

You've already seen (above) that to get an event immediate when an element is interacted with in some way you set the `enable_events` parameter.

```
import PySimpleGUI as sg

"""
    Restrict the characters allowed in an input element to digits and . or -
    Accomplished by removing last character input if not a valid character
"""

layout = [  [sg.Text('Input only floating point numbers')],
            [sg.Input(key='-IN-', enable_events=True)],
            [sg.Button('Exit')]  ]

window = sg.Window('Floating point input validation', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    # if last character in input element is invalid, remove it
    if event == '-IN-' and values['-IN-'] and values['-IN-'][-1] not in ('0123456789.-'):
        window['-IN-'].update(values['-IN-'][:-1])
window.close()
```

This code only allows entry of the correct characters.

Note that this example does not fully validate that the entry is a valid floating point number, but rather that it has the correct characters.

If you wanted to take it a step further and verify that the entry is actually a valid floating point number, then you can change the "if" statement to test for valid floating point number.

```
import PySimpleGUI as sg

"""
    Restrict the characters allowed in an input element to digits and . or -
    Accomplished by removing last character input if not a valid character
"""

layout = [  [sg.Text('Input only floating point numbers')],
            [sg.Input(key='-IN-', enable_events=True)],
            [sg.Button('Exit')]  ]

window = sg.Window('Floating point input validation', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    # if last character in input element is invalid, remove it
    if event == '-IN-' and values['-IN-']:
        try:
            in_as_float = float(values['-IN-'])
        except:
            if len(values['-IN-']) == 1 and values['-IN-'][0] == '-':
                continue
            window['-IN-'].update(values['-IN-'][:-1])
window.close()
```

Recipe - Positioning Windows on a Multi-Monitor Setup (tkinter version of PySimpleGUI only)

On the Windows operating system, it's possible to create your window on monitors other than your primary display. Think of your primary display as a single quadrant in a larger space of display area. The upper left corner of your primary display is (0,0).

If you wish to locate / create a window on the monitor to the LEFT of your primary monitor, then set the X value to a **negative** value. This causes the window to be created on the monitor to the left. If you set your X value to be larger than the width of your primary monitor, then you window will be created on the monitor that is located to the RIGHT of your primary monitor.]

I don't know if this technique works on Linux, but it's working great on Windows. This technique has been tried on a 4-monitor setup and it worked as you would expect.

To use this feature, rather than using the default window location of "centered on your primary screen", set the `location` parameter in your `Window` creation to be the location you wish the window to be created.

Setting the parameter `location=(-500,330)` in my `Window` call, set the location of the window on my left hand monitor.

Experimenting is the best way to get a handle on how your system responds.

It would be great to know if this works on Linux and the Mac.

This code is from a Demo Program named `Demo_Window_Location_Finder.py` and will help you located the x,y position on your monitors. Grab the yellow square with your mouse to move the tool around your screen. The 4 arrows point to the direction indicated


```
import PySimpleGUI as sg

sg.theme('dark green 7')

layout = [
    [sg.T(sg.SYMBOL_UP_ARROWHEAD),
     sg.Text(size=(None, 1), key='-OUT-'),
     sg.Text(size=(None, 1), key='-OUT2-', expand_x=True, expand_y=True, justification='c'), sg.T(
    [sg.T('Screen size: '), sg.T(sg.Window.get_screen_size()), sg.T(sg.SYMBOL_SQUARE)],
    [sg.T(sg.SYMBOL_DOWN_ARROWHEAD),
     sg.Text(size=(None, 1), key='-OUT4-'),
     sg.Text(size=(None, 1), key='-OUT3-', expand_x=True, expand_y=True, justification='r'), sg.T(
]

window = sg.Window('Title not seen', layout, grab_anywhere=True, no_titlebar=True, margins=(0, 0),
                   keep_on_top=True, font='_ 25', finalize=True, transparent_color=sg.theme_backgr

while True:
    event, values = window.read(timeout=100)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == 'Edit Me':
        sg.execute_editor(__file__)

    loc = window.current_location()
    window['-OUT-'].update(loc)
    window['-OUT2-'].update((loc[0] + window.size[0], loc[1]))
    window['-OUT3-'].update((loc[0] + window.size[0], loc[1] + window.size[1]))
    window['-OUT4-'].update((loc[0], loc[1] + window.size[1]))

window.close()
```

Recipe - Element Justification and Alignment

There are 2 terms used in PySimpleGUI regarding positioning: * Justification - Positioning on the horizontal axis (left, center, right) * Alignment - Positioning on the vertical axis (top, middle, bottom)

Justification

Justification of elements can be accomplished using 2 methods.
1. Use a Column Element with the `element_justification` parameter 2. Use the `Push` element

The `Push` was added in 2021 to the tkinter port. The PySimpleGUIQt port already has an element called `Stretch` that works in a similar way. You could say that `Push` is an alias for `Stretch`, even though `Stretch` wasn't a tkinter element previously.

Push

The way to think about `Push` elements is to think of them a an element that "repels" or pushes around other elements. The `Push` works on a row by row basis. Each row that you want the `Push` to impact will need to have one or more `Push` elements on that row.

Normally, each row is left justified in PySimpleGUI (unless you've set a parameter in the Window object or the row is in a Column element that has a setting that impacts justification.

Think of the sides of a window as a wall that cannot move. Elements can move, but the side-walls cannot. If you place a `Push` element on the left side of another element, then it will "push" the element to the right. If you place a `Push` on the right side, then it will "push" the element to the left. If you use TWO `Push` elements and place one on each side of an element, then the element will be centered.

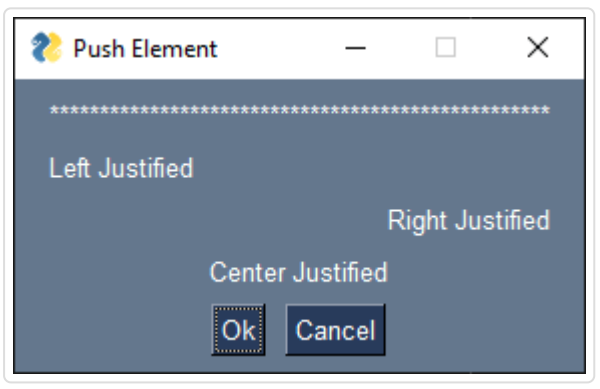
This recipe demonstrates using a `Push` element to create rows that have different justification happening on each row.

The first row of the layout is 50 chars so that the window will be wide enough that each row's justification will have some room to move around.

The second row doesn't **need** a `Push` element in order for the element to be left justified. However, if your entire window was right justified, then using the `Push` on the right side of an element would push it to be left justified.

Notice the last row of the layout. There are 2 buttons together with a Push on each side. This causes those 2 buttons to be centered.

```
import PySimpleGUI as sg

layout = [[sg.Text('*'*50)],
          [sg.Text('Left Justified'), sg.Push()],
          [sg.Push(), sg.Text('Right Justified')],
          [sg.Push(), sg.Text('Center Justified'), sg.Push()],
          [sg.Push(), sg.Button('Ok'), sg.Button('Cancel'), sg.Push()]]

window = sg.Window('Push Element', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break

window.close()
```

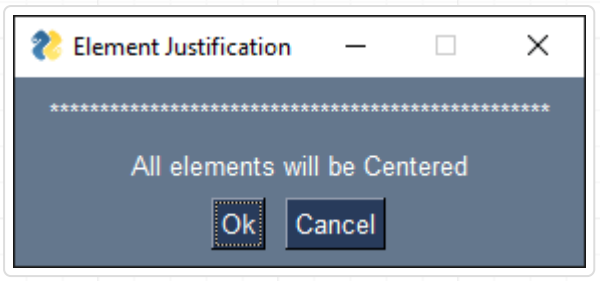


Container Element Justification (`element_justification` parameter)

You can use Container Elements (Column, Frame, Tab and Window too) to justify multiple rows at a time. The parameter `element_justification` controls how elements within a container or Window are justified.

In this example, all elements in the window are centered

```
import PySimpleGUI as sg

layout = [[sg.Text('*'*50)],
          [sg.Text('All elements will be Centered')],
          [sg.Button('Ok'), sg.Button('Cancel')]]

window = sg.Window('Element Justification', layout, element_justification='c')

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break

window.close()
```



Alignment - single rows

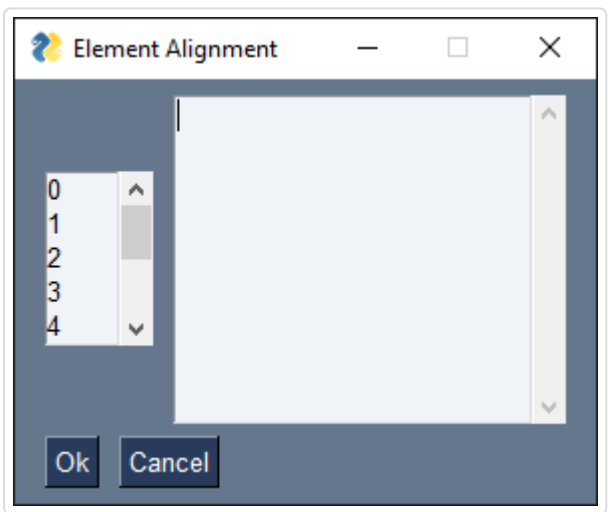
"Alignment" is the term used to describe the vertical positioning of elements. Within a single row, alignment is performed by using a container element or by using a one of the alignment "layout helper functions". You'll find the layout helper functions in the call reference documentation here: https://pysimplegui.readthedocs.io/en/latest/call%20reference/#layout-helper-funcs

There are 3 functions in particular that affect vertical positioning:
* vtop - Align an element or an entire row to the "top" of the row * vbottom - Align an element or an entire row to the "bottom" of the row * vcenter - Align an element or an entire row to the "center" of the row

By default the alignment on each row is center.

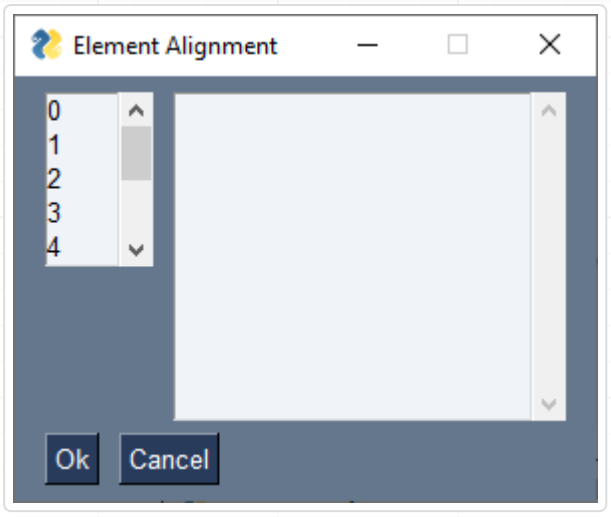
This program uses the default alignment which will center elements on each row.

```
import PySimpleGUI as sg

layout = [[sg.Listbox(list(range(10)), size=(5,5)), sg.Multiline(size=(25,10))],
          [sg.Button('Ok'), sg.Button('Cancel')]]

window = sg.Window('Element Alignment', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break

window.close()
```



If you want to have the Listbox and the Multiline aligned at the top, then you can use the `vtop` helper function. Since the Listbox is the shorter height, you could add `vtop` just to that element. Or, you can use `vtop` and pass in the entire row so that if the size of one of these elements changes in the future so that the Multiline is shorter, they'll remain top aligned.

If the elements on that row were top-aligned, the window will look like this:



Here are 3 ways you can accomplish this operation.

Align only the single element

```
import PySimpleGUI as sg

layout = [[sg.vtop(sg.Listbox(list(range(10)), size=(5,5))), sg.Multiline(size=(25,10))],
          [sg.Button('Ok'), sg.Button('Cancel')]]

window = sg.Window('Element Alignment', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break

window.close()
```

Align the entire row

```
import PySimpleGUI as sg

layout = [sg.vtop([sg.Listbox(list(range(10)), size=(5,5)), sg.Multiline(size=(25,10))]),
          [sg.Button('Ok'), sg.Button('Cancel')]]

window = sg.Window('Element Alignment', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break

window.close()
```

Notice how using `vtop` in the example above replaces the entire row with just the `vtop` call. The reason for this is that the `vtop` function returns a list (i.e. a row). This means that brackets are not needed. But it looks a little odd and makes it more difficult to see where the rows are.

Newer versions of PySimpleGUI allow an extra set of brackets `[ ]` so that the layout appears to still be a list-per-row.

```
import PySimpleGUI as sg

layout = [[sg.vtop([sg.Listbox(list(range(10)), size=(5,5)), sg.Multiline(size=(25,10))])],
          [sg.Button('Ok'), sg.Button('Cancel')]]

window = sg.Window('Element Alignment', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break

window.close()
```

Alignment - `VPush`

Just like the `Push` element will "push" elements around in a horizontal fashion, the `VPush` element pushes entire groups of rows up and down within the container they are inside of.

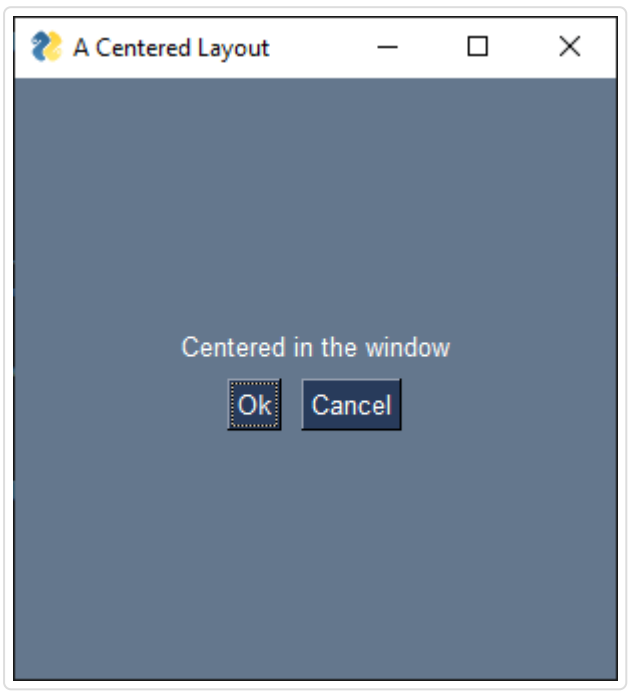
If you have a single `VPush` in your layout, then the layout will be pushed to the top or to the bottom. Normally layouts are top-aligned by default so there's no need to have a single `VPush` at the bottom. If you have two `VPush` elements, then it will center the elements between them.

One of the best examples of using `VPush` is when a window's size has been hard coded. Hard coding a window's size is **not recommended** in PySimpleGUI. The reason is that the contents inside may not fit inside your hard coded size on some computers. It's usually better to allow the window's size to "float" and be automatically sized to fit the contents.

Perhaps a better example would be if you wanted to allow your window to be resized and have the contents vertically aligned after resizing.

But, if you're determined to hard code a size and want to vertically center your elements in that window, then the `VPush` is a good way to go.

This example window is 300 pixels by 300 pixels. The layout is both center justified and center aligned. This is accomplished using a combination of `Push` and `VPush` elements.




```
import PySimpleGUI as sg

layout = [[sg.VPush()],
          [sg.Push(), sg.Text('Centered in the window'), sg.Push()],
          [sg.Push(), sg.Button('Ok'), sg.Button('Cancel'), sg.Push()],
          [sg.VPush()]]

window = sg.Window('A Centered Layout', layout, resizable=True, size=(300, 300))

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break
```

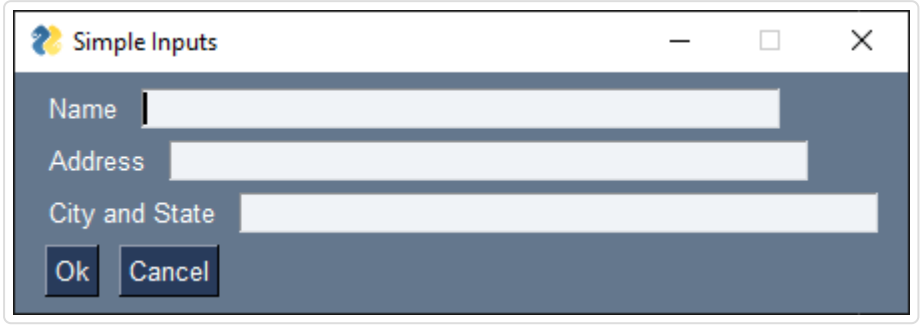
Recipe - Clean Simple Inputs

Single Line Inputs

Many of our first GUIs involve collecting small bits of information and then continuing on with processing that input. They are simple programs that are quick for beginners to knock out and get a few accomplishments so their confidence builds.

The most basic of these are layouts that have a `Text` Element and an `Input` Element. This is a basic "form" the user fills out. Maybe you've got code that looks like this already.

```
import PySimpleGUI as sg

layout = [  [sg.Text('Name'), sg.Input(key='-NAME-')],
            [sg.Text('Address'), sg.Input(key='-ADDRESS-')],
            [sg.Text('City and State'), sg.Input(key='-CITY AND STATE-')],
            [sg.Ok(), sg.Cancel()]]

window = sg.Window('Simple Inputs', layout)

while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Cancel':
        break

window.close()
```



`Push` Your `Input` Rows

One easy addition to your layout is to "push" each input row to the right. The effect is quite nice and the implementation involves one simple operation - add a `Push` element to the start of each row that has an `Input`

The change looks like this:

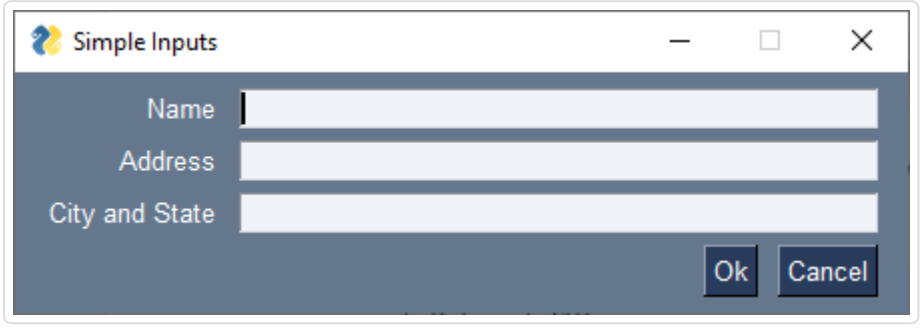
```
import PySimpleGUI as sg

layout = [  [sg.Push(), sg.Text('Name'), sg.Input(key='-NAME-')],
            [sg.Push(), sg.Text('Address'), sg.Input(key='-ADDRESS-')],
            [sg.Push(), sg.Text('City and State'), sg.Input(key='-CITY AND STATE-')],
            [sg.Ok(), sg.Cancel()]]

window = sg.Window('Simple Inputs', layout)

while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Cancel':
        break

window.close()
```

And the result is surprisingly pleasant.

Push Everything - `element_justification='r'`

If you really want to get clever and save time as well, you can make a 1-parameter change to your `Window` definition and get a similar result. The one difference is that the buttons will also get pushed over.

```
import PySimpleGUI as sg

layout = [  [sg.Text('Name'), sg.Input(key='-NAME-')],
            [sg.Text('Address'), sg.Input(key='-ADDRESS-')],
            [sg.Text('City and State'), sg.Input(key='-CITY AND STATE-')],
            [sg.Ok(), sg.Cancel()]]

window = sg.Window('Simple Inputs', layout, element_justification='r')

while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Cancel':
        break

window.close()
```



Recipe - Printing

Outputting text is a very common operation in programming. Your first Python program may have been

```
print('Hello World')
```

But in the world of GUIs where do "prints" fit in? Well, lots of places! Of course you can still use the normal `print` statement. It will output to StdOut (standard out) which is normally the shell where the program was launched from.

Printing to the console becomes a problem however when you launch using `pythonw` on Windows or if you launch your program in some other way that doesn't have a console. With PySimpleGUI you have many options available to you so fear not.

These Recipes explore how to retain *prints already in your code*. Let's say your code was written for a console and you want to migrate over to a GUI. Maybe there are so many print statements that you don't want to modify every one of them individually.

There are **at least 3 ways** to transform your `print` statements that we'll explore here 1. The Debug window 2. The Output Element 3. The Multiline Element

The various forms of "print" you'll be introduced to all support the `sep` and `end` parameters that you find on normal print statements.

Recipe Printing - #1/4 Printing to Debug Window

The debug window acts like a virtual console. There are 2 operating modes for the debug window. One re-routes stdout to the window, the other does not.

`Print` - Print to the Debug Window

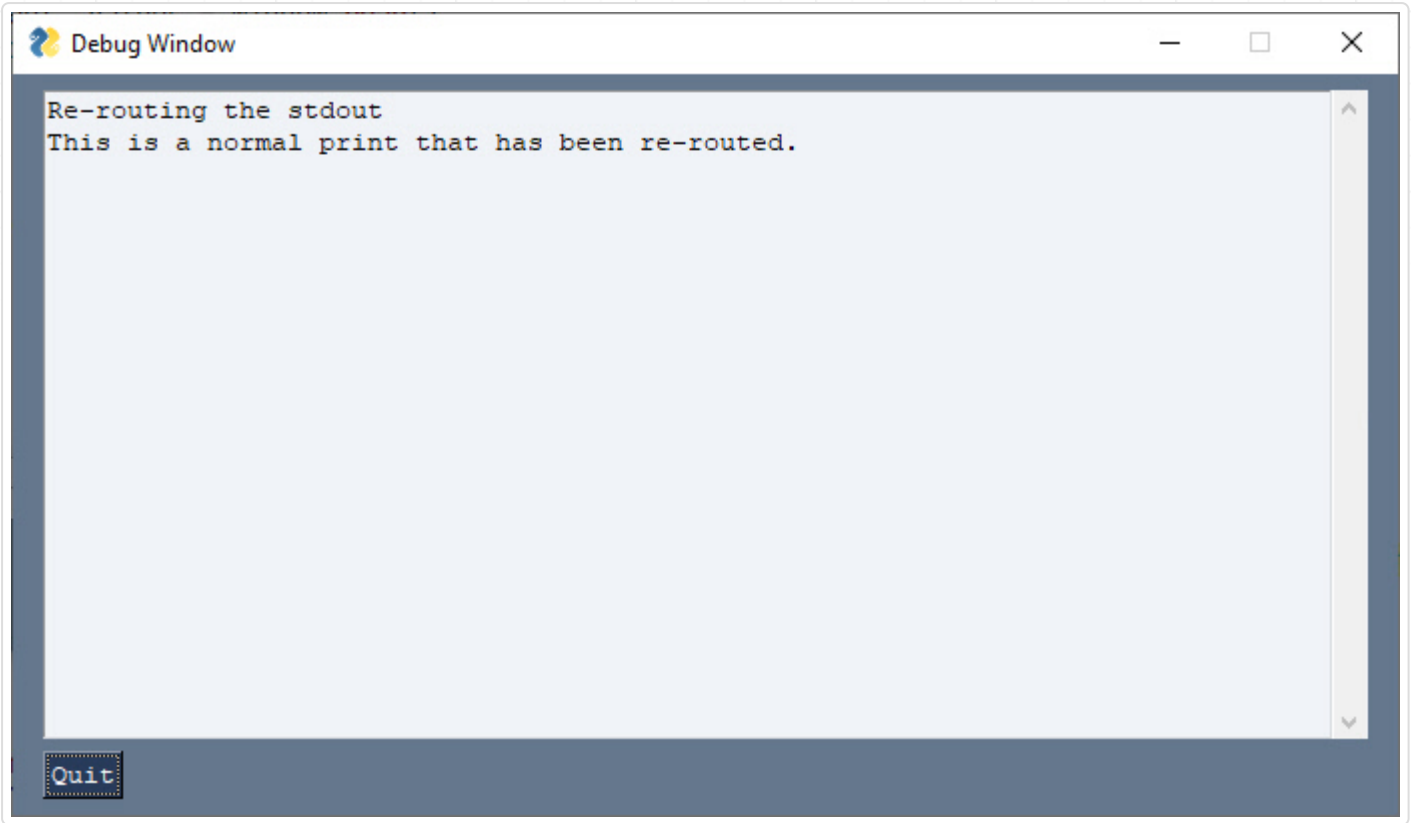
The functions `Print`, `eprint`, `EasyPrint` all refer to the same funtion. There is no difference whic hyou use as they point to identical code. The one you'll see used in Demo Programs is `Print`.

One method for routing your print statements to the debuyg window is to reassign the `print` keyword to be the PySimpleGUI function `Print`. This can be done through simple assignment.

`print = sg.Print`

You can also remap stdout to the debug window by calling `Print` with the parameter `do_not_reroute_stdout = False`. This will reroute all of your print statements out to the debug window.

```
import PySimpleGUI as sg

sg.Print('Re-routing the stdout', do_not_reroute_stdout=False)
print('This is a normal print that has been re-routed.')
```

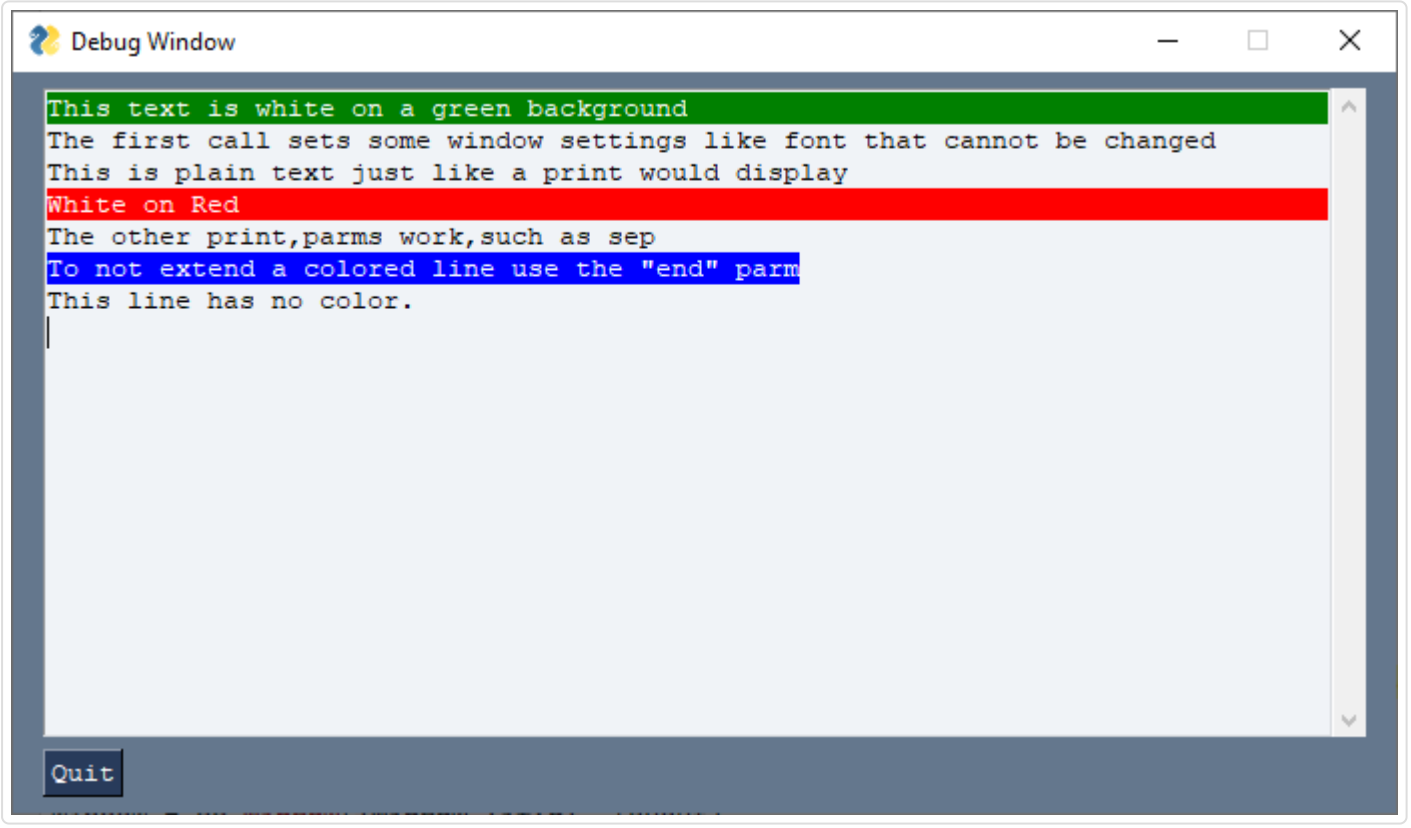


While both `print` and `sg.Print` will output text to your Debug Window.

Printing in color is only operational if you do not reroute stdout to the debug window.

If color printing is important, then don't reroute your stdout to the debug window. Only use calls to `Print` without any change to the stdout settings and you'll be able to print in color.

```
import PySimpleGUI as sg

sg.Print('This text is white on a green background', text_color='white', background_color='green',
sg.Print('The first call sets some window settings like font that cannot be changed')
sg.Print('This is plain text just like a print would display')
sg.Print('White on Red', background_color='red', text_color='white')
sg.Print('The other print', 'parms work', 'such as sep', sep=',')
sg.Print('To not extend a colored line use the "end" parm', background_color='blue', text_color='w
sg.Print('\nThis line has no color.')
```



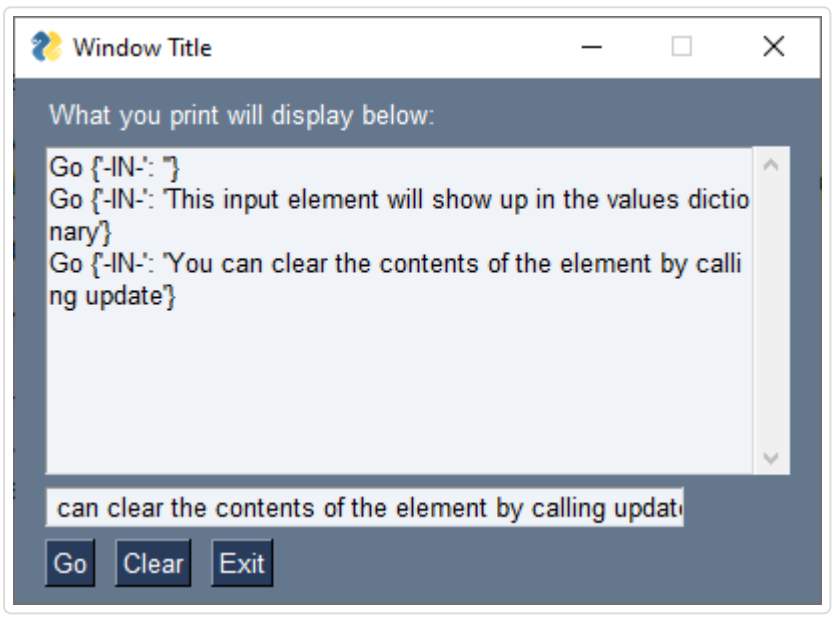
Recipe Printing - #2/4 Print to `Output` Element

If you want to re-route your standard out to your window, then placing an `Output` Element in your layout will do just that. When you call "print", your text will be routed to that `Output` Element. Note you can only have 1 of these in your layout because there's only 1 stdout.

Of all of the "print" techniques, this is the best to use if you cannot change your print statements. The `Output` element is the best choice if your prints are in another module that you don't have control over such that "redefining / reassigning" what `print` does isn't a possibility.

This layout with an `Output` element shows the results of a few clicks of the Go Button.

```
import PySimpleGUI as sg

layout = [  [sg.Text('What you print will display below:')],
            [sg.Output(size=(50,10), key='-OUTPUT-')],
            [sg.In(key='-IN-')],
            [sg.Button('Go'), sg.Button('Clear'), sg.Button('Exit')]  ]

window = sg.Window('Window Title', layout)

while True:             # Event Loop
    event, values = window.read()
    print(event, values)
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    if event == 'Clear':
        window['-OUTPUT-'].update('')
window.close()
```



Recipe Printing - #3/4 Print to `Multiline` Element

Beginning in 4.18.0 you can "print" to any `Multiline` Element in your layouts. The `Multiline.print` method acts similar to the `Print` function described earlier. It has the normal print parameters `sep` & `end` and also has color options. It's like a super-charged `print` statement.

"Converting" exprint print statements to output to a `Multiline` Element can be done by either

- Adding the `Multiline` element to the `print` statment so that it's calling the `Multiline.print` method
- Redefining `print`

Added in version 4.25.0 was the ability to re-route stdout and stderr directly to any `Multiline` element. This is done using parameteres when you create the multiline or you can call class methods to do the rerouting operation after the element is created.

Since you may not be able to always have access to the window when printing, especially in code that it not your own code, another parameter was added `auto_refresh`. If set to True then the window will automatically refresh every time an update is made to that Multiline element.

3A Appending Element to `print` Statement to print to Multiline

Let's try the first option, adding the element onto the front of an existing `print` statement as well as using the color parameters.

The most basic form of converting your exiting `print` into a `Multline` based `print` is to add the same element-lookup code that you would use when calling an element's `update` method. Generically, that conversion looks like this:

```
print('Testing 1 2 3')
```

If our Multiline's key is '-ML-' then the expression to look the element up is:

```
window['-ML-']
```

Combing the two transforms the original print to a `Multline` element print:

```
window['-ML-'].print('Testing 1 2 3')
```

Because we're using these `Multilne` elements as output only elements, we don't want to have their contents returned in the values dictionary when we call `window.read()`. To make any element not be included in the values dictionary, add the constant `WRITE_ONLY_KEY` onto the end of your key. This would change our previous example to:

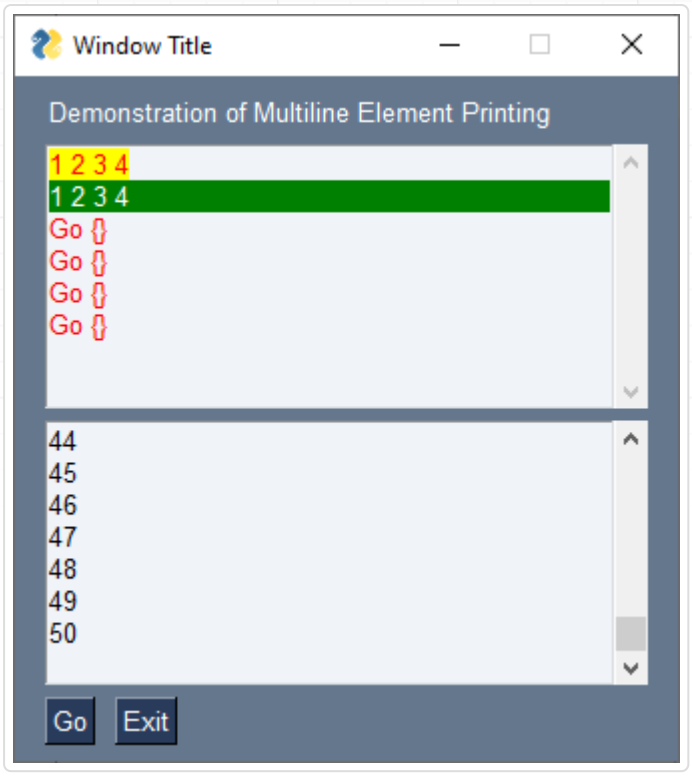
```
window['-ML-'+sg.WRITE_ONLY_KEY].print('Testing 1 2 3')
```

When you define the multiline element in your layout, its key will need to have this suffix added too.

Combining all of this information into a full-program we arrive at this Recipe:

```
import PySimpleGUI as sg

layout = [  [sg.Text('Demonstration of Multiline Element Printing')],
            [sg.MLine(key='-ML1-'+sg.WRITE_ONLY_KEY, size=(40,8))],
            [sg.MLine(key='-ML2-'+sg.WRITE_ONLY_KEY,  size=(40,8))],
            [sg.Button('Go'), sg.Button('Exit')]]

window = sg.Window('Window Title', layout, finalize=True)


# Note, need to finalize the window above if want to do these prior to calling window.read()
window['-ML1-'+sg.WRITE_ONLY_KEY].print(1,2,3,4,end='', text_color='red', background_color='yellow
window['-ML1-'+sg.WRITE_ONLY_KEY].print('\n', end='')
window['-ML1-'+sg.WRITE_ONLY_KEY].print(1,2,3,4,text_color='white', background_color='green')
counter = 0

while True:             # Event Loop
    event, values = window.read(timeout=100)
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    if event == 'Go':
        window['-ML1-'+sg.WRITE_ONLY_KEY].print(event, values, text_color='red')
    window['-ML2-'+sg.WRITE_ONLY_KEY].print(counter)
    counter += 1
window.close()
```

It produces this window:



There are a number of tricks and techniques burried in this Recpie so study it closely as there are a lot of options being used.

3B Redefining `print` to Print to `Multiline`

If you want to use the `Multline` element as the destination for your print, but you don't want to go through your code and modify every print statement by adding an element lookup, then you can simply redefine your call to `print` to either be a function that adds that multline element onto the print for you or a lambda expression if you want to make it a single line of code. Yes, it's not suggested to use a lambda expression by assignment to a vairable, but sometimes it may be easier to understand. Find the right balanace for you and ryour projct.

If you were to use a funciton, then your code my look like this:

```
def mprint(*args, **kwargs):
    window['-ML1-' + sg.WRITE_ONLY_KEY].print(*args, **kwargs)

print = mprint
```

A named lambda expression would perhaps resemeble this:

```
print = lambda *args, **kwargs: window['-ML1-' + sg.WRITE_ONLY_KEY].print(*args, **kwargs)
```

Putting it all together into a single block of code for you to copy and run results in

```
def mprint(*args, **kwargs):
    window['-ML1-'+sg.WRITE_ONLY_KEY].print(*args, **kwargs)

print = mprint

# Optionally could use this lambda instead of the mprint function
# print = lambda *args, **kwargs: window['-ML1-' + sg.WRITE_ONLY_KEY].print(*args, **kwargs)

layout = [  [sg.Text('Demonstration of Multiline Element Printing')],
            [sg.MLine(key='-ML1-'+sg.WRITE_ONLY_KEY, size=(40,8))],
            [sg.MLine(key='-ML2-'+sg.WRITE_ONLY_KEY,  size=(40,8))],
            [sg.Button('Go'), sg.Button('Exit')]]

window = sg.Window('Window Title', layout, finalize=True)
print(1,2,3,4,end='', text_color='red', background_color='yellow')
print('\n', end='')
print(1,2,3,4,text_color='white', background_color='green')
counter = 0

# Switch to printing to second multiline
print = lambda *args, **kwargs: window['-ML2-' + sg.WRITE_ONLY_KEY].print(*args, **kwargs)


while True:             # Event Loop
    event, values = window.read(timeout=100)
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    if event == 'Go':
        print(event, values, text_color='red')
    print(counter)
    counter += 1
window.close()
```

Recipe 3C - Rerouting stdout and stderr directly to a Multiline

This was made available to the tkinter port in version 4.25.0.

The eaiest way to make this happen is using parmaters when creating the `Multline` Element

- reroute_stdout
- reroute_stderr

If you wish to reroute stdout / stderr after you've already created (and finalized) the Multline, then you can call `reroute_stdout_to_here` to reroute stdeout and `reroute_stderr_to_here` to reroute stderr.

To restore the old values back, be sure and call `restore_stdout` and `restore_stderr`

This has a risky component to this.

Warning Regarding Threading and Printing

If programs outside of your control are running threads and they happen to call print, then the stdout will be routed to the window. This MAY cause tkinter to crash.

Your thread, by calling print, will trigger code inside of PySimpleGUI itself to be executed. This code can be significant if the stdout has been re-rerouted to a multiline element that has auto-refresh turned on for example. It is unclean how many operations or queued or if the calls from the threads will directly impact tkinter.

The point here it to simple be on the looking for the dreaded "tkinter not in the mainloop" error

Recipe Printing - #4A/4 using `cprint` function (color printing) to print to Multiline

This method was added to PySimpleGUI tkinter port in June 2020 and needs to be ported to the other ports still.

The idea is have a function, `cprint` that looks and acts like a normal print.... except, you can "route" it to any multiline element. There are 2 ways to do routing.

1. Call `cprint_set_output_destination(window, multiline_key)` to tell PySimpleGUI where the output should go
2. Indicate the output location directly in the `cprint` call itself

Color

The color portion of the cprint call is achieved through additional parameters that are not normally present on a call to print. This means that if you use these color parameters, you cannot simply rename your `cprint` calls to be `print` calls. Of course you can safely go the other direction, renaming your `print` calls to call `cprint`.

The Aliasing Shortcut Trick

While covered in "cprint", this trick can save you MASSIVE amount of typing. It works well in PyCharm too.

Would I do this in a huge production code base. No, but I'm wring a little 100 line packet of fun.

IF you're tire of writine `sg.xxxxx` as much as I am, then maybe you'll like this hack too.

Through simple assignment, you can rename PySimpleGUI functions. You add them to your local name space so you no longer need the `sgl.` part.

For example. I'm tired of writing `sg.cprint`. I can fix this by adding a line of code to make an alias and then using the aliase insteast.

I could even make it super short.

```
cp = sg.cprint
```

Now all I call is `cp('This is what I want to print')` The cool thing about PyCharm is that it knows these are the same and so the DOCSTRINGS work with them!!

Yes, you can rename the entire Elements ane still get all the documentation as you type it in.

So that you don't have to type: `sg.cprint` every time you want to print, you can add this statement to the top of your code:

`cprint = sg.cprint`

Now you can simply call `cprint` directly. You will still get the docstrings if you're running PyCharm so you're not losing anything there.

Recipe 4A

This Recipe shows many of the concepts and parameters. There is also one located in the Demo Programs area on GitHub (http://Demos.PySimpleGUI.org).

```
import PySimpleGUI as sg

"""
    Demo - cprint usage

    "Print" to any Multiline Element in any of your windows.

    cprint in a really handy way to "print" to any multiline element in any one of your windows.
    There is an initial call - cprint_set_output_destination, where you set the output window and
    for the Multiline Element.

    There are FOUR different ways to indicate the color, from verbose to the most minimal are:
    1. Specify text_color and background_color in the cprint call
    2. Specify t, b paramters when calling cprint
    3. Specify c/colors parameter a tuple with (text color, background color)
    4. Specify c/colors parameter as a string "text on background"  e.g.  "white on red"

    Copyright 2020 PySimpleGUI.org
"""

def main():
    cprint = sg.cprint

    MLINE_KEY = '-ML-'+sg.WRITE_ONLY_KEY   # multiline element's key. Indicate it's an output only
    MLINE_KEY2 = '-ML2-'+sg.WRITE_ONLY_KEY # multiline element's key. Indicate it's an output only

    output_key = MLINE_KEY

    layout = [  [sg.Text('Multiline Color Print Demo', font='Any 18')],
                [sg.Multiline('Multiline\n', size=(80,20), key=MLINE_KEY)],
                [sg.Multiline('Multiline2\n', size=(80,20), key=MLINE_KEY2)],
                [sg.Text('Text color:'), sg.Input(size=(12,1), key='-TEXT COLOR-'),
                 sg.Text('on Background color:'), sg.Input(size=(12,1), key='-BG COLOR-')],
                [sg.Input('Type text to output here', size=(80,1), key='-IN-')],
                [sg.Button('Print', bind_return_key=True), sg.Button('Print short'),
                 sg.Button('Force 1'), sg.Button('Force 2'),
                 sg.Button('Use Input for colors'), sg.Button('Toggle Output Location'), sg.Button

    window = sg.Window('Window Title', layout)

    sg.cprint_set_output_destination(window, output_key)

    while True:             # Event Loop
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Exit':
            break
        if event == 'Print':
            cprint(values['-IN-'], text_color=values['-TEXT COLOR-'], background_color=values['-BG
        elif event == 'Print short':
            cprint(values['-IN-'], c=(values['-TEXT COLOR-'], values['-BG COLOR-']))
        elif event.startswith('Use Input'):
            cprint(values['-IN-'], colors=values['-IN-'])
        elif event.startswith('Toggle'):
            output_key = MLINE_KEY if output_key == MLINE_KEY2 else MLINE_KEY2
            sg.cprint_set_output_destination(window, output_key)
            cprint('Switched to this output element', c='white on red')
        elif event == 'Force 1':
            cprint(values['-IN-'], c=(values['-TEXT COLOR-'], values['-BG COLOR-']), key=MLINE_KEY
        elif event == 'Force 2':
            cprint(values['-IN-'], c=(values['-TEXT COLOR-'], values['-BG COLOR-']), key=MLINE_KEY
    window.close()

if __name__ == '__main__':
    main()
```

Recipe Printing - #4B/4 using `cprint` with Multiline Parameters (PySimpleGUI version 4.25.0+)

Beginning in verison 4.25.0 of the tkinter port you'll find new parameters for the Multline Element that makes the job of re-routihn your output much easier. Rather than calling the `cprint_set_output_destination` function, you will use the `Multline` element's initial parameters to both setup the routing of the print output, but also mark the element as being a write-only element. You can set the parameter `write_only` to True in order to make this a write-only Multiline.

The new parameters you'll be interested in are:

- write_only

- auto_refresh
- reroute_cprint

This will cut out the call previously required to set up the routing. You will be setting up the routing through the Multiline creation ifself.

You will continue to be able to manually route stdout and stderr to the Multline uning the `reroute_stdout_to_here` call. Sorry about the wordiness of the call, but you're probably only going to have one in your code. So it didn't seem so bad to have something descriptive enough that you won't need a comment.

Automatic Refresh

The Multiline element has an option for auto-refreshing after an update. The Output element automatically refreshes after each write. Hopefully this will not slow things down considerably.

Here is the code for 4B

```
import threading
import time
import PySimpleGUI as sg


"""
    Threaded Demo - Uses Window.write_event_value communications

    Requires PySimpleGUI.py version 4.25.0 and later

    This is a really important demo  to understand if you're going to be using multithreading in P

    Older mechanisms for multi-threading in PySimpleGUI relied on polling of a queue. The manageme
    queue is now performed internally to PySimpleGUI.

    The importance of using the new window.write_event_value call cannot be emphasized enough.  It
    a positive way, on your code to move to this mechanism as your code will simply "pend" waiting

    Copyright 2020 PySimpleGUI.org
"""

THREAD_EVENT = '-THREAD-'

cp = sg.cprint

def the_thread(window):
    """
    The thread that communicates with the application through the window's events.

    Once a second wakes and sends a new event and associated value to the window
    """
    i = 0
    while True:
        time.sleep(1)
        window.write_event_value('-THREAD-', (threading.current_thread().name, i))      # Data sen
        cp('This is cheating from the thread', c='white on green')
        i += 1


def main():
    """
    The demo will display in the multiline info about the event and values dictionary as it is bei
    returned from window.read()
    Every time "Start" is clicked a new thread is started
    Try clicking "Dummy" to see that the window is active while the thread stuff is happening in t
    """

    layout = [  [sg.Text('Output Area - cprint\'s route to here', font='Any 15')],
                [sg.Multiline(size=(65,20), key='-ML-', autoscroll=True, reroute_stdout=True, writ
                [sg.T('Input so you can see data in your dictionary')],
                [sg.Input(key='-IN-', size=(30,1))],
                [sg.B('Start A Thread'), sg.B('Dummy'), sg.Button('Exit')]  ]

    window = sg.Window('Window Title', layout)

    while True:             # Event Loop
        event, values = window.read()
        cp(event, values)
        if event == sg.WIN_CLOSED or event == 'Exit':
            break
        if event.startswith('Start'):
            threading.Thread(target=the_thread, args=(window,), daemon=True).start()
        if event == THREAD_EVENT:
            cp(f'Data from the thread ', colors='white on purple', end='')
            cp(f'{values[THREAD_EVENT]}', colors='white on red')
    window.close()


if __name__ == '__main__':
    main()
```

Recipe - Save and Load Program Settings

There is an entire set of API calls now available to you in PySimpleGUI to help with "settings".

Please check out the demo programs as there are numerous demos that have calls to the settings APIs.

There is also a section in the main documentation about these APIs. They are detailed in the call reference as well.

The basics are that in your layout or before your layout, you'll read your settings. If your program changes any settings, they will immediately be saved to your settings file. You never have to worry about loading or saving the settings file. It happens automatically. Just start calling the get and set calls if using the "function interface".

If you do not set a filename, then the name of your .py or .pyw file will be used. Beware, however, that turning your program into an EXE can impact the settings file as your filename won't look the same to the Python code. If you're going to turn your code into an EXE, then it's best to explicitly set the filename.

This statement sets the filename. Note that the path is not indicated in this example. PySimpleGUI will store all the settings in the default settings folder if you don't specify one.

```
sg.user_settings_filename(filename='DaysUntil.json')
```

Reading a setting can be as easy as this call:

```
theme = sg.user_settings_get_entry('-theme-', 'Dark Gray 13')
```

The first parm is the "key" and the second is the default value if no setting is found.

Saving a setting can be done with this call:

```
sg.user_settings_set_entry('-theme-', my_new_theme)
```

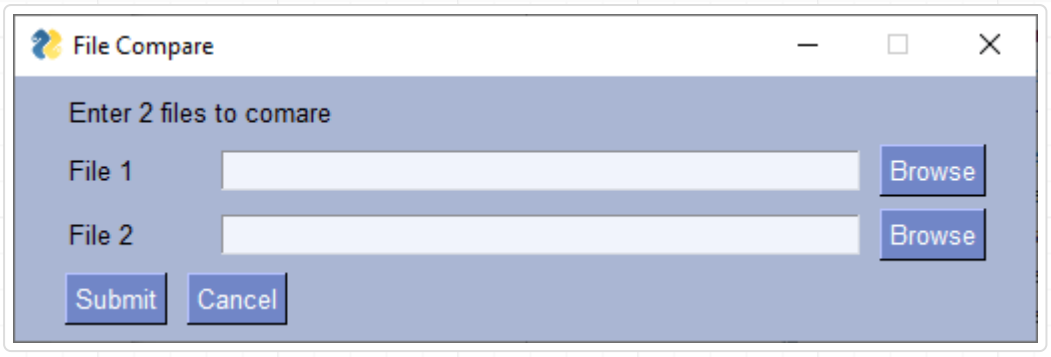
Take a look at the main documentation for the Object interface if you would prefer it over the function based interface. These API calls are used in any demo program that has settings and for all PySimpleGUI projects that have settings. They're super SIMPLE to use and work well.

A Simple Save Filename Example

```
import PySimpleGUI as sg

def save_previous_filename_demo():
    """
    Saving the previously selected filename....
    A demo of one of the likely most popular use of user settings
    * Use previous input as default for Input
    * When a new filename is chosen, write the filename to user settings
    """

    # Notice that the Input element has a default value given (first parameter) that is read from
    layout = [[sg.Text('Enter a filename:')],
              [sg.Input(sg.user_settings_get_entry('-filename-', ''), key='-IN-'), sg.FileBrowse()
              [sg.B('Save'), sg.B('Exit Without Saving', key='Exit'), sg.T('(or click X to close w

    window = sg.Window('Filename Example', layout)

    while True:
        event, values = window.read()
        if event in (sg.WINDOW_CLOSED, 'Exit'):
            break
        elif event == 'Save':
            sg.user_settings_set_entry('-filename-', values['-IN-'])

    window.close()


if __name__ == '__main__':
    sg.user_settings_filename(path='.')  # Set the location for the settings file
    # Run a couple of demo windows
    save_previous_filename_demo()
```

Recipe - Get 2 Files By Browsing

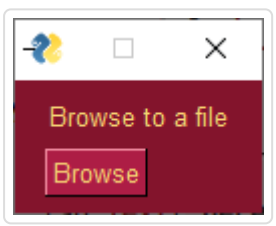
Sometimes you just need to get a couple of filenames. Browse to get 2 file names that can be then compared. By using `Input` elements the user can either use the Browse button to browse to select a file or they can paste the filename into the input element directly.




```
import PySimpleGUI as sg

sg.theme('Light Blue 2')

layout = [[sg.Text('Enter 2 files to comare')],
          [sg.Text('File 1', size=(8, 1)), sg.Input(), sg.FileBrowse()],
          [sg.Text('File 2', size=(8, 1)), sg.Input(), sg.FileBrowse()],
          [sg.Submit(), sg.Cancel()]]

window = sg.Window('File Compare', layout)

event, values = window.read()
window.close()
print(f'You clicked {event}')
print(f'You chose filenames {values[0]} and {values[1]}')
```

This pattern is really good any time you've got a file or folder to get from the user. By pairing an `Input` element with a browse button, you give the user the ability to do a quick paste if they've already got the path on the clipboard or they can click "Browse" and browse to get the filename/foldername.

Recipe - Get Filename With No Input Display. Returns when file selected



There are times when you don't want to display the file that's chosen and you want the program to start when the user chooses a file. One way of doing this is to hide the input field that's filled in by the "Browse Button". By enabling events for the input field, you'll get an event when that field is filled in.

```
import PySimpleGUI as sg

sg.theme('Dark Red')

layout = [[sg.Text('Browse to a file')],
          [sg.Input(key='-FILE-', visible=False, enable_events=True), sg.FileBrowse()]]

event, values = sg.Window('File Compare', layout).read(close=True)

print(f'You chose: {values["-FILE-"]}')
```

Recipe - Long Operations - Multi-threading

IMPORTANT GUI Topic!

Brief summary:

PySimpleGUI can help you with running long operations as threads without you needing to learn the threading library. The `Window` method `perform_long_operation` makes this serious GUI problem a non-issue.

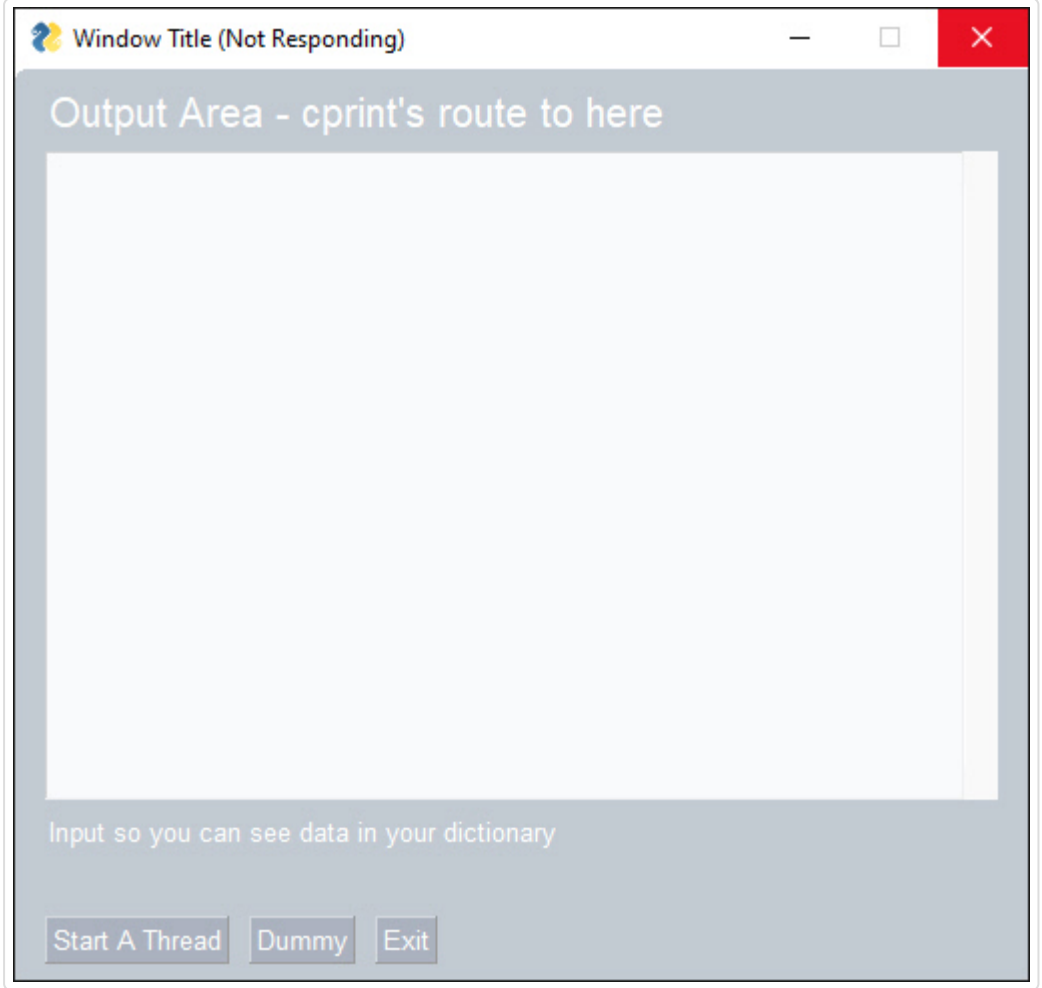
Threads can "inject" events and data into a `window.read()` call. This allows your GUI application to simply stop, pend and awaken immediately when something happens. This makes for zero CPU time used when northing's happening and it means 0ms latency. In other words, you're not polling, you're pending.

The Long Operation

A classic problem of GUI programming is when you try to perform some operation that requires a lot of time. The problem is simple enough.... you have a GUI and when you press a button, you want a 10 second operation to take place while you're GUI patiently waits.

What happens to most people that give this a try gets the dreaded Windows/Linux/Mac "Your program has stopped responding do you wish to close it"

If you add a sleep(30) to your code, it's not very many seconds before your window does this:



No Bueno

PySimpleGUI `Window.perform_long_operation`

To get you over the initial hump of multi-threaded programming, you can let PySimpleGUI create and manage threads for you. Like other APIs in PySimpleGUI, it's been simplified down as far as possible.

Here's the basic steps using `perform_long_operation` 1. Pass your function name and a key to the call to `window.perform_long_operation` 2. Continue running your GUI event loop 3. Windows pend using their typical `window.read()` call 4. You will get the event when your function returns 5. The `values` dictionary will contain your function's return value. They key will be the same as the event. So, `values[event]` is your function's return value.

```
import PySimpleGUI as sg
import time

# My function that takes a long time to do...
def my_long_operation():
    time.sleep(15)
    return 'My return value'


def main():
    layout = [  [sg.Text('My Window')],
                [sg.Input(key='-IN-')],
                [sg.Text(key='-OUT-')],
                [sg.Button('Go'), sg.Button('Threaded'), sg.Button('Dummy')]  ]

    window = sg.Window('Window Title', layout, keep_on_top=True)

    while True:             # Event Loop
        event, values = window.read()
        if event == sg.WIN_CLOSED:
            break

        window['-OUT-'].update(f'{event, values}')  # show the event and values in the window
        window.refresh()                            # make sure it's shown immediately

        if event == 'Go':
            return_value = my_long_operation()
            window['-OUT-'].update(f'direct return value = {return_value}')
        elif event  == 'Threaded':
            # Let PySimpleGUI do the threading for you...
            window.perform_long_operation(my_long_operation, '-OPERATION DONE-')
        elif event  == '-OPERATION DONE-':
            window['-OUT-'].update(f'indirect return value = {values[event]}')

    window.close()

if __name__ == '__main__':
    main()
```

What if my function takes parameters?

Note that the first parameter for `perform_long_operation` is your function. If you're like most of us, you'll enter `my_func()` instead of `my_func`. The first actually calls your function immediately, the second passes your function's object rather than calling it.

If you need to pass parameters to your function, then you'll need to make one simple change... add a lambda. Think of it as how you would want your function called.

In the Demo Program for this call, `Demo_Long_Operations.py`, it uses a function that takes parameters as the example. Here is the line of code from that demo:

```
# This is where the magic happens.  Add your function call as a lambda
window.perform_long_operation(lambda :
                              my_long_func(int(values['-IN-']), a=10),
                              '-END KEY-')
```

I've broken the code up with newlines to emphasize where your function call goes. A more common format may be:

```
window.perform_long_operation(lambda : my_long_func(int(values['-IN-']), a=10), '-END KEY-')
```

Here is the function definition that is to be called:

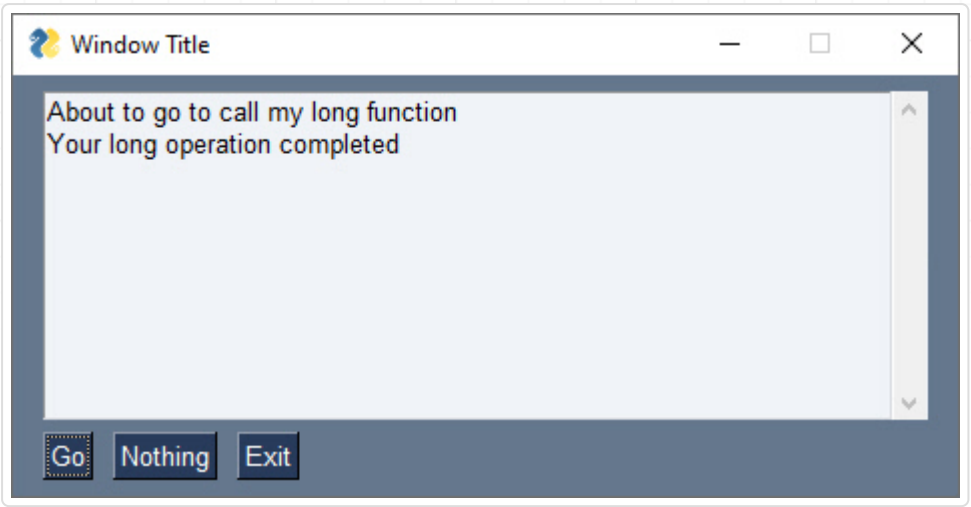
```
def my_long_func(count, a=1, b=2):
```

The Thread-based Solution

If you're ready to jump on into threading, then you can do that too.

Here's the basic steps 1. You put your long-running operation into a thread 2. Your thread signals the window when it is done 3. Windows pend using their typical `window.read()` call 4. The `values` dictionary will contain your function's return value if you pass it through

Take a moment to get to know the code. You'll find the typical event loop. If you run this program, and you don't touch anything like your mouse, then it should sit for 10 seconds doing nothing and then print out the completed the message.



If you attempted to interact with the window by pressing the "Nothing" button, then you will likely get a message about your window stopped responding.

Threaded Long Operation

I think we can agree that brute force, no matter how badly we want it to work, won't. Bummer

```
import PySimpleGUI as sg
import time
import threading


def long_function_thread(window):
    time.sleep(10)
    window.write_event_value('-THREAD DONE-', '')

def long_function():
    threading.Thread(target=long_function_thread, args=(window,), daemon=True).start()


layout = [[sg.Output(size=(60,10))],
          [sg.Button('Go'), sg.Button('Nothing'), sg.Button('Exit')]  ]

window = sg.Window('Window Title', layout)

while True:             # Event Loop
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == 'Go':
        print('About to go to call my long function')
        long_function()
        print('Long function has returned from starting')
    elif event == '-THREAD DONE-':
        print('Your long operation completed')
    else:
        print(event, values)
window.close()
```

If you click the "Nothing" button, then you'll get a line printed in the Multiline that has the event and the values dictionary.

Because there are no "input" elements, your values dictionary is empty.

Clicking "Go" is when the fun begins.

You are immediately shown a message that the long-operation function is starting. The same function name as before is called `long_function`. But now the contents of that function have been replaced with starting a thread that executes the same code.

This single line of code is all that was needed to create our long0runing function as a thread and to start that thread:

```
threading.Thread(target=the_thread, args=(window,), daemon=True).start()
```

The conversion over to a thread was done in 3 simple steps:

1. Renamed the `long_fundtion` to `long_function_thread`
2. Pass into the `long_function_thread` the `window` that it will communicate with
3. Add call to `window.write_event_value` when the long_running_thread is existing

The result is a GUI that continues to operate and be responsive to user's requests during the long running operation.

Long operations with feedback

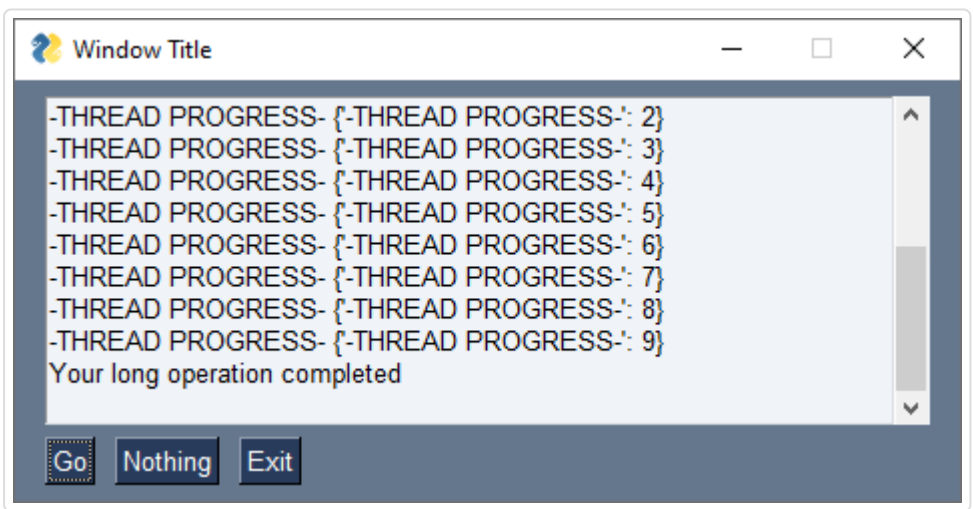
The power of the `Window.write_event_value` is that it can be used at any time, not just at the beginning and end of operations. If a long operation can be broken into smaller parts, then progress can be shown to the user. Rather than calling `Window.write_event_value` one time, it can be called a number of times too.

If we modify the code so that instead of sleeping for 10 seconds, we sleep for 1 second 10 times, then it's possible to show information about progress.

Here's the code with the new operation broken up into 10 parts

```
import PySimpleGUI as sg
import time
import threading

def long_function_thread(window):
    for i in range(10):
        time.sleep(1)
        window.write_event_value('-THREAD PROGRESS-', i)
    window.write_event_value('-THREAD DONE-', '')

def long_function():
    threading.Thread(target=long_function_thread, args=(window,), daemon=True).start()


layout = [[sg.Output(size=(60,10))],
          [sg.Button('Go'), sg.Button('Nothing'), sg.Button('Exit')]  ]

window = sg.Window('Window Title', layout)

while True:             # Event Loop
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == 'Go':
        print('About to go to call my long function')
        long_function()
        print('Long function has returned from starting')
    elif event == '-THREAD DONE-':
        print('Your long operation completed')
    else:
        print(event, values)
window.close()
```

And the resulting window



Recipe - convert_to_bytes Function + PIL Image Viewer

This function has turned out to be one of the best for working with images in PySimpleGUI.

```
import PIL.Image
import io
import base64

def convert_to_bytes(file_or_bytes, resize=None):
    '''
    Will convert into bytes and optionally resize an image that is a file or a base64 bytes object
    Turns into  PNG format in the process so that can be displayed by tkinter
    :param file_or_bytes: either a string filename or a bytes base64 image object
    :type file_or_bytes:  (Union[str, bytes])
    :param resize:  optional new size
    :type resize: (Tuple[int, int] or None)
    :return: (bytes) a byte-string object
    :rtype: (bytes)
    '''
    if isinstance(file_or_bytes, str):
        img = PIL.Image.open(file_or_bytes)
    else:
        try:
            img = PIL.Image.open(io.BytesIO(base64.b64decode(file_or_bytes)))
        except Exception as e:
            dataBytesIO = io.BytesIO(file_or_bytes)
            img = PIL.Image.open(dataBytesIO)

    cur_width, cur_height = img.size
    if resize:
        new_width, new_height = resize
        scale = min(new_height/cur_height, new_width/cur_width)
        img = img.resize((int(cur_width*scale), int(cur_height*scale)), PIL.Image.ANTIALIAS)
    bio = io.BytesIO()
    img.save(bio, format="PNG")
    del img
    return bio.getvalue()
```

It requires 3 packages - PIL, io, and base64. PIL is the only one you'll need to pip install.

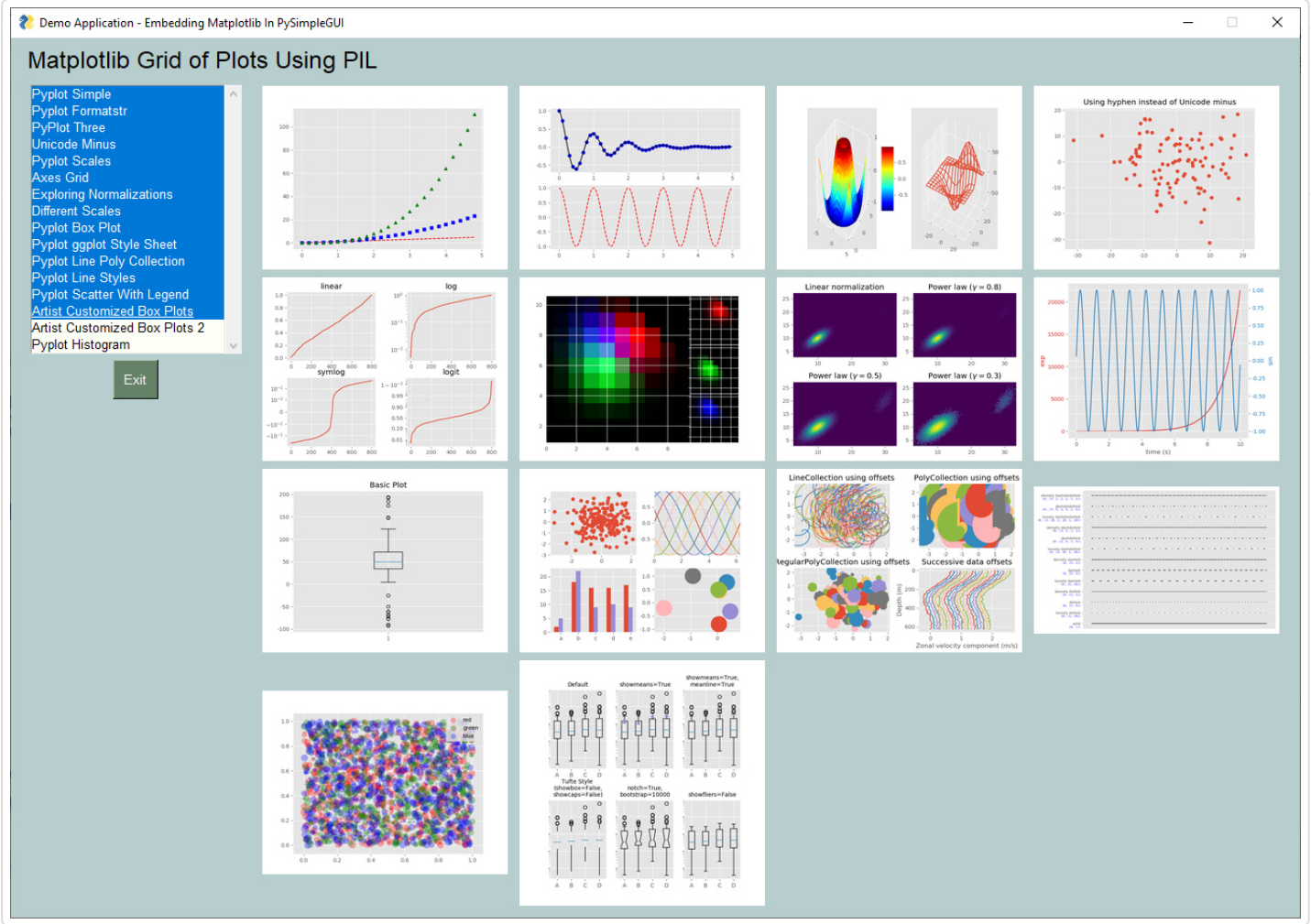
PySimpleGUI does not directly use the PIL package so that PySimpleGUI can remain highly portable. Requiring users to install PIL was simply not acceptable for the package, but it's fine for demo programs and helper functions like this one.

One thing that PIL buys you is the ability to work with a LOT more file formats. If you want JPG images, then you want to use PIL as the tkinter based PySimpleGUI only supports PNGs and GIFs (because that's all tht tkinter supports)

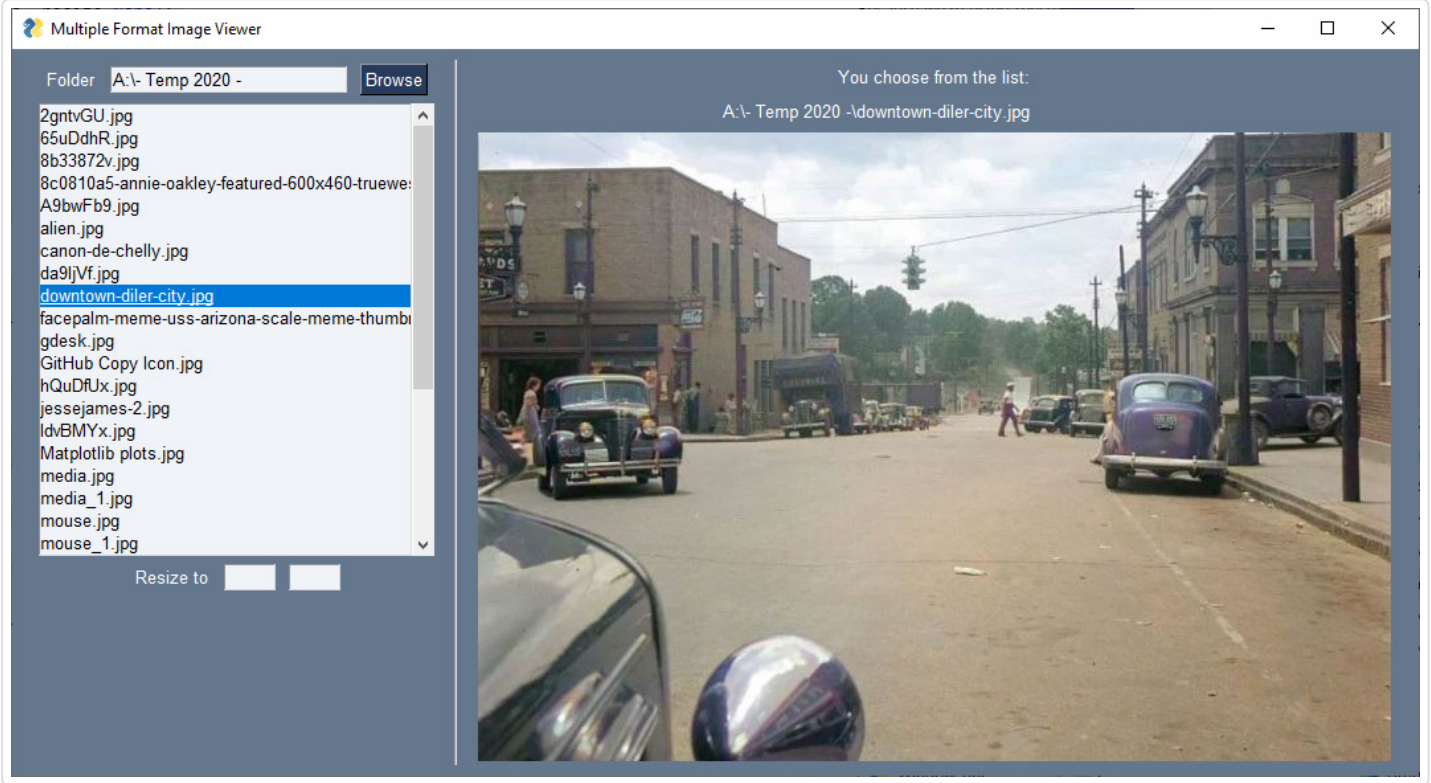
`convert_to_types` is a fantastic little function because you can give it a filename or a bytes string and it will return a bytes string that is optionally resized.

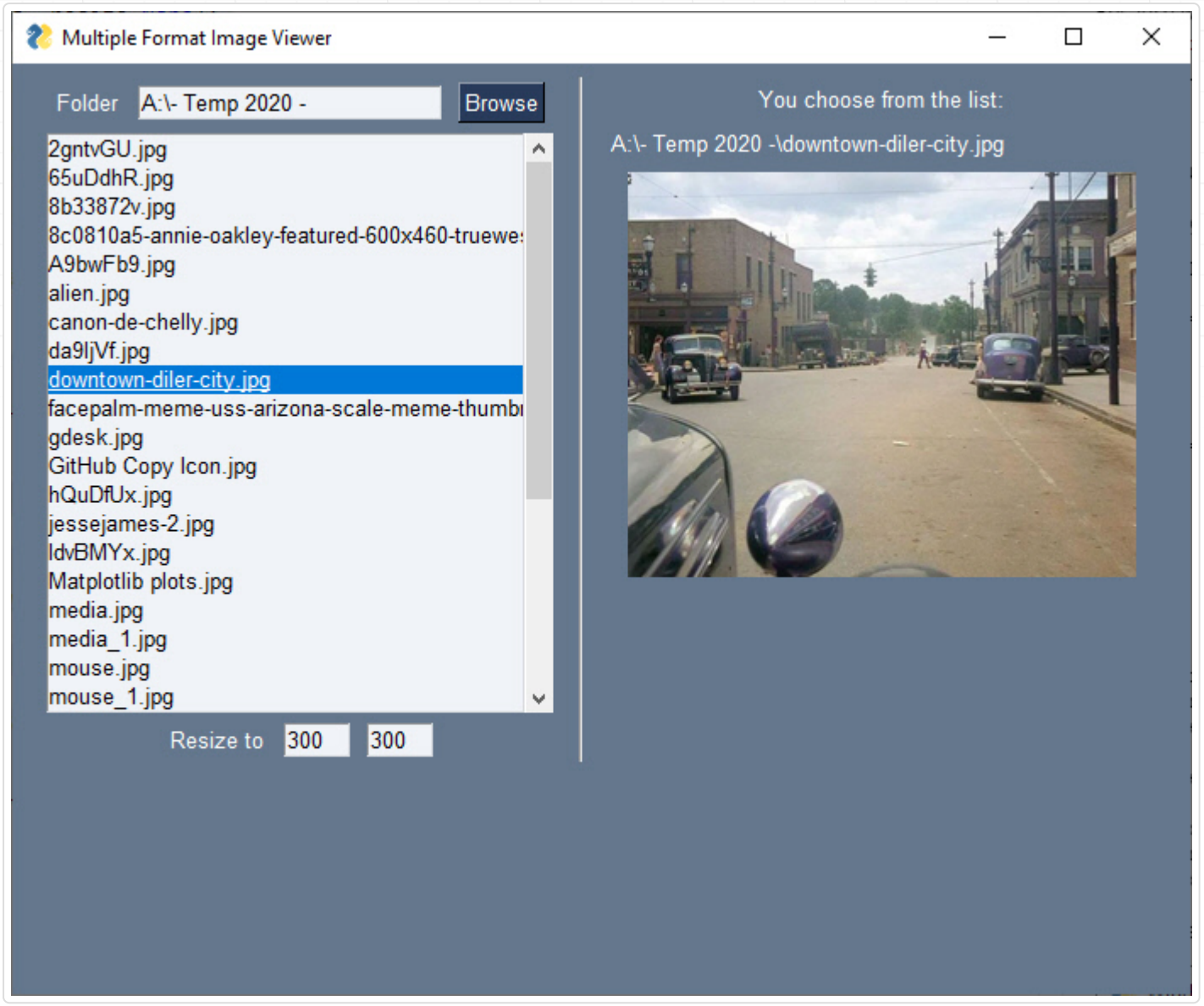
This makes working with button images MUCH MUCH easier.

Here are a couple of demos that use this function. The first one is a Matplotlib previewer. It creates a grid of graphs



These shots are from the demo that you'll see the source code to below. Note the "Resize to" field below the file list.





As an example of one way to use this function, included here is the Demo Program you'll find in the demo programs area on the GitHub:

```
import PySimpleGUI as sg
# import PySimpleGUIQt as sg
import os.path
import PIL.Image
import io
import base64

"""
    Demo for displaying any format of image file.

    Normally tkinter only wants PNG and GIF files.  This program uses PIL to convert files
    such as jpg files into a PNG format so that tkinter can use it.

    The key to the program is the function "convert_to_bytes" which takes a filename or a
    bytes object and converts (with optional resize) into a PNG formatted bytes object that
    can then be passed to an Image Element's update method.  This function can also optionally
    resize the image.

    Copyright 2020 PySimpleGUI.org
"""



def convert_to_bytes(file_or_bytes, resize=None):
    '''
    Will convert into bytes and optionally resize an image that is a file or a base64 bytes object
    Turns into  PNG format in the process so that can be displayed by tkinter
    :param file_or_bytes: either a string filename or a bytes base64 image object
    :type file_or_bytes:  (Union[str, bytes])
    :param resize:  optional new size
    :type resize: (Tuple[int, int] or None)
    :return: (bytes) a byte-string object
    :rtype: (bytes)
    '''
    if isinstance(file_or_bytes, str):
        img = PIL.Image.open(file_or_bytes)
    else:
        try:
            img = PIL.Image.open(io.BytesIO(base64.b64decode(file_or_bytes)))
        except Exception as e:
            dataBytesIO = io.BytesIO(file_or_bytes)
            img = PIL.Image.open(dataBytesIO)

    cur_width, cur_height = img.size
    if resize:
        new_width, new_height = resize
        scale = min(new_height/cur_height, new_width/cur_width)
        img = img.resize((int(cur_width*scale), int(cur_height*scale)), PIL.Image.ANTIALIAS)
    bio = io.BytesIO()
    img.save(bio, format="PNG")
    del img
    return bio.getvalue()



# --------------------------------- Define Layout ---------------------------------

# First the window layout...2 columns

left_col = [[sg.Text('Folder'), sg.In(size=(25,1), enable_events=True ,key='-FOLDER-'), sg.FolderB
            [sg.Listbox(values=[], enable_events=True, size=(40,20),key='-FILE LIST-')],
            [sg.Text('Resize to'), sg.In(key='-W-', size=(5,1)), sg.In(key='-H-', size=(5,1))]]

# For now will only show the name of the file that was chosen
images_col = [[sg.Text('You choose from the list:')],
              [sg.Text(size=(40,1), key='-TOUT-')],
              [sg.Image(key='-IMAGE-')]]

# ----- Full layout -----
layout = [[sg.Column(left_col, element_justification='c'), sg.VSeperator(),sg.Column(images_col, e

# --------------------------------- Create Window ---------------------------------
window = sg.Window('Multiple Format Image Viewer', layout,resizable=True)

# ----- Run the Event Loop -----
# --------------------------------- Event Loop ---------------------------------
while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == '-FOLDER-':                         # Folder name was filled in, make a list of fi
```

```
        folder = values['-FOLDER-']
        try:
            file_list = os.listdir(folder)         # get list of files in folder
        except:
            file_list = []
        fnames = [f for f in file_list if os.path.isfile(
            os.path.join(folder, f)) and f.lower().endswith((".png", ".jpg", "jpeg", ".tiff", ".bm
        window['-FILE LIST-'].update(fnames)
    elif event == '-FILE LIST-':    # A file was chosen from the listbox
        try:
            filename = os.path.join(values['-FOLDER-'], values['-FILE LIST-'][0])
            window['-TOUT-'].update(filename)
            if values['-W-'] and values['-H-']:
                new_size = int(values['-W-']), int(values['-H-'])
            else:
                new_size = None
            window['-IMAGE-'].update(data=convert_to_bytes(filename, resize=new_size))
        except Exception as E:
            print(f'** Error {E} **')
            pass        # something weird happened making the full filename
# --------------------------------- Close & Exit ---------------------------------
window.close()
```

Use with buttons

One particuarly good use of this function is when you want to add a graphic to a button. This function will convert images of any format into a byte string that can be passed into your Button element when you create it.

Let's say you want to user the PySimpleGUI icon as a button. You can do that easily enough using this statement:

```
sg.Button(image_data=sg.DEFAULT_BASE64_ICON, button_color=(sg.theme_background_color(), sg.theme_b
```

But maybe your application has button images that are all 40 x 40 pixels. In that case, you simply pass this image to the convert function along with the new size you want it to be.

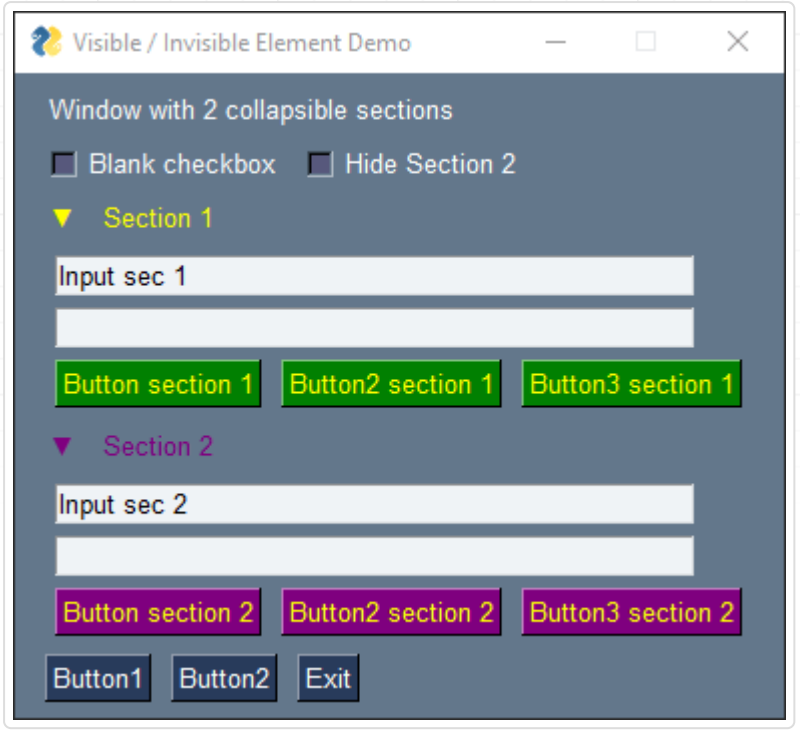
```
sg.Button(image_data=convert_to_bytes(sg.DEFAULT_BASE64_ICON, (40, 40)), button_color=(sg.theme_ba
```

The result are these 2 buttons:

Recipe - Collapsible Sections (Visible / Invisible Elements)

Setting elements to be invisible and visible again has been a big challenge until version 4.28.0 of the tkinter port of PySimpleGUI. This is when the `pin` function was added. This function will "pin" an element to a location in the layout. Without this pin, then the element may move when made inivisible and visible again. There was also a problem of the space not shrinking when make from visible to invisible. With the `pin` function, this problem was solved. There is a small, 1 pixel, cost to this operation. When the element is invisible, there will be a single pixel where it is pinned. These may add up if you have 40+ rows of invisible elements. This is not typical however so it tends to work pretty well.

This recipe shows how to use invisible elements to create a window with 2 sections that can be collapsed down to a single line with an arrow on it. You could just as easily make the entire section totally disappear if you wanted. In other words, you're not limited to using invisible elements in this way only.



You will also find this program in the Demo Programs section on GitHub.

```
import PySimpleGUI as sg

"""
    Demo - "Collapsible" sections of windows

    This demo shows one techinique for creating a collapsible section (Column) within your window.

    It uses the "pin" function so you'll need version 4.28.0+

    A number of "shortcut aliases" are used in the layouts to compact things a bit.
    In case you've not encountered these shortcuts, the meaning are:
    B = Button, T = Text, I = Input = InputText, k = key
    Also, both methods for specifying Button colors were used (tuple / single string)
    Section #2 uses them the most to show you what it's like to use more compact names.

    To open/close a section, click on the arrow or name of the section.
    Section 2 can also be controlled using the checkbox at the top of the window just to
    show that there are multiple way to trigger events such as these.

    Copyright 2020 PySimpleGUI.org
"""


SYMBOL_UP =    '▲'
SYMBOL_DOWN =  '▼'


def collapse(layout, key):
    """
    Helper function that creates a Column that can be later made hidden, thus appearing "collapsed
    :param layout: The layout for the section
    :param key: Key used to make this seciton visible / invisible
    :return: A pinned column that can be placed directly into your layout
    :rtype: sg.pin
    """
    return sg.pin(sg.Column(layout, key=key))


section1 = [[sg.Input('Input sec 1', key='-IN1-')],
            [sg.Input(key='-IN11-')],
            [sg.Button('Button section 1',  button_color='yellow on green'),
             sg.Button('Button2 section 1', button_color='yellow on green'),
             sg.Button('Button3 section 1', button_color='yellow on green')]]

section2 = [[sg.I('Input sec 2', k='-IN2-')],
            [sg.I(k='-IN21-')],
            [sg.B('Button section 2',  button_color=('yellow', 'purple')),
             sg.B('Button2 section 2', button_color=('yellow', 'purple')),
             sg.B('Button3 section 2', button_color=('yellow', 'purple'))]]


layout =   [[sg.Text('Window with 2 collapsible sections')],
            [sg.Checkbox('Blank checkbox'), sg.Checkbox('Hide Section 2', enable_events=True, key=
            #### Section 1 part ####
            [sg.T(SYMBOL_DOWN, enable_events=True, k='-OPEN SEC1-', text_color='yellow'), sg.T('Se
            [collapse(section1, '-SEC1-')],
            #### Section 2 part ####
            [sg.T(SYMBOL_DOWN, enable_events=True, k='-OPEN SEC2-', text_color='purple'),
             sg.T('Section 2', enable_events=True, text_color='purple', k='-OPEN SEC2-TEXT')],
            [collapse(section2, '-SEC2-')],
            #### Buttons at bottom ####
            [sg.Button('Button1'),sg.Button('Button2'), sg.Button('Exit')]]

window = sg.Window('Visible / Invisible Element Demo', layout)

opened1, opened2 = True, True

while True:             # Event Loop
    event, values = window.read()
    print(event, values)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break

    if event.startswith('-OPEN SEC1-'):
        opened1 = not opened1
        window['-OPEN SEC1-'].update(SYMBOL_DOWN if opened1 else SYMBOL_UP)
        window['-SEC1-'].update(visible=opened1)

    if event.startswith('-OPEN SEC2-'):
        opened2 = not opened2
        window['-OPEN SEC2-'].update(SYMBOL_DOWN if opened2 else SYMBOL_UP)
        window['-OPEN SEC2-CHECKBOX'].update(not opened2)
        window['-SEC2-'].update(visible=opened2)
```

```
window.close()
```

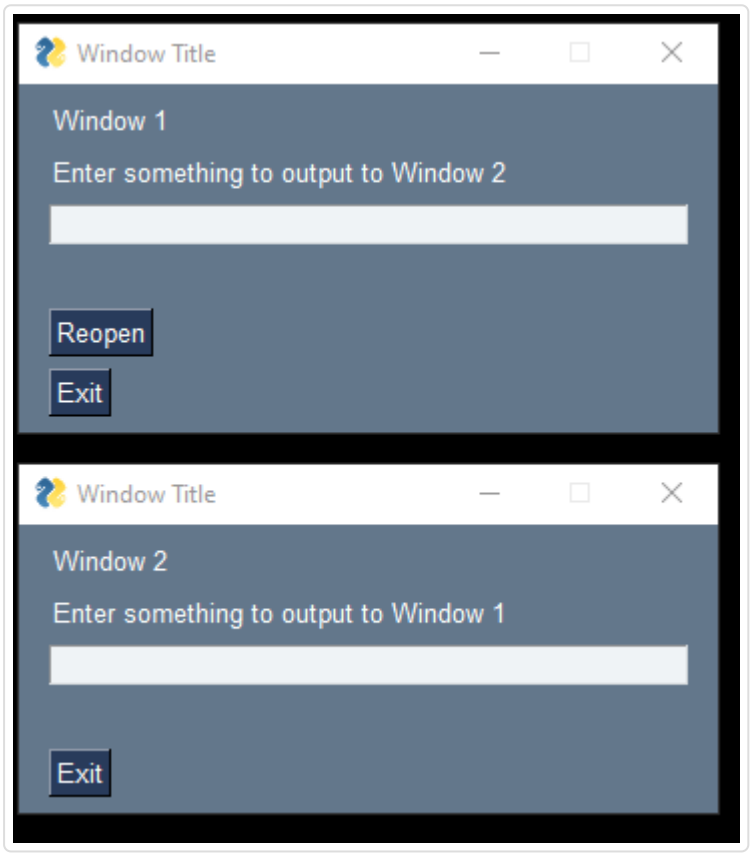
Recipe Multiple Windows - `read_all_windows`

Beginning in version 4.28.0 you'll find that working with multiple windows in the tkinter port of PySimpleGUI to be much much easier.

This Recipe shows 2 windows. Both of them are active and can be interacted with. When you enter something in window 1 it is updated in window 2. Notice that the keys are named the same in both windows. This makes it really easy to write generic code that will update fields in either window, the only difference will be which Window is updated.

You'll find that you'll have less chances for problems like "reusing layouts" if you put your layout and window creation into a function. This will guarantee a "fresh" window every time you call the function. If you close window 2 and then click the "Reopen" button in window 1, then all that is needed is to call the `make_win2` function again and move the new window to the location below the first window.

The program remains active until both windows have been closed.

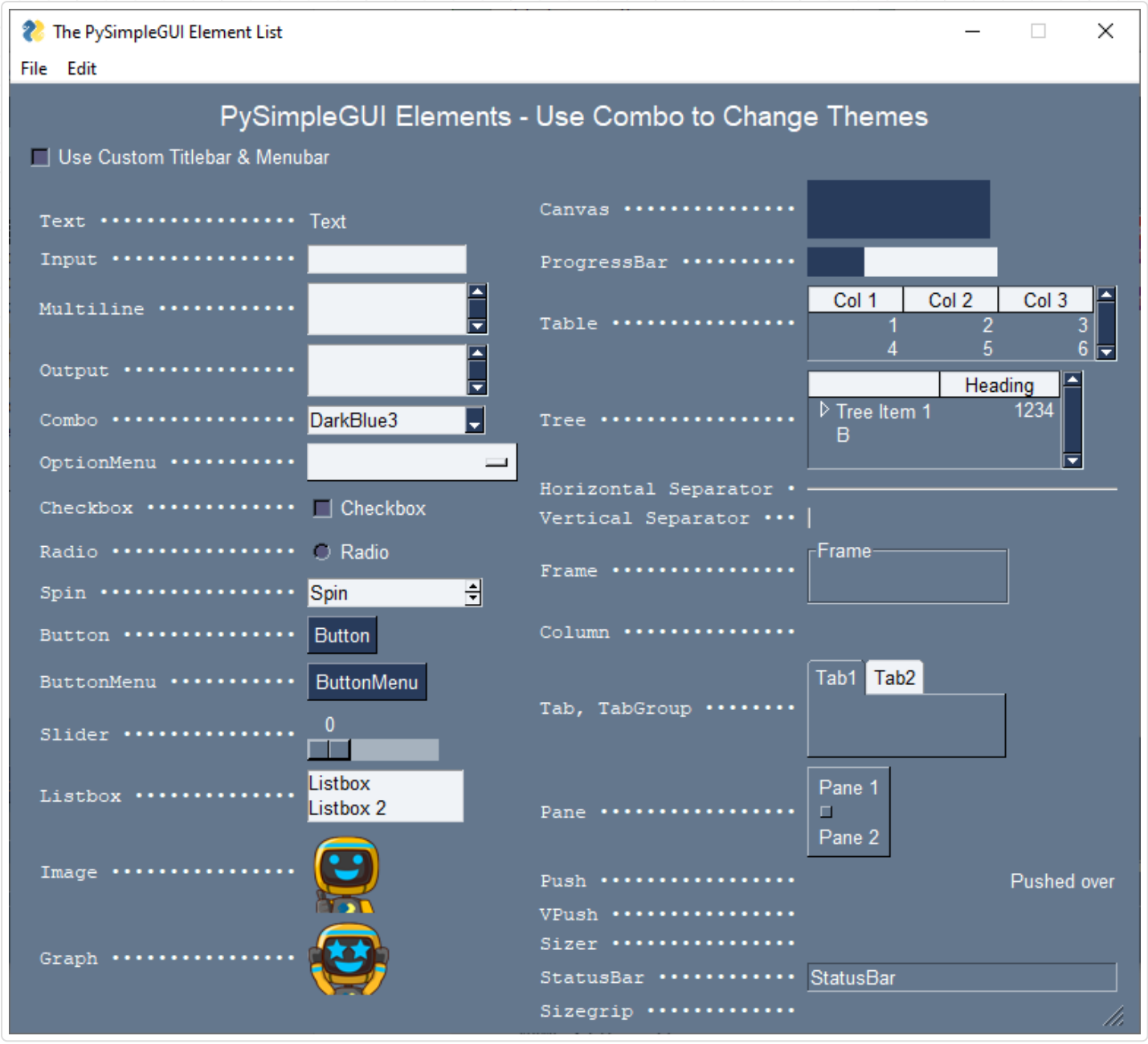


```
import PySimpleGUI as sg
"""
    Demo - 2 simultaneous windows using read_all_window

    Both windows are immediately visible.  Each window updates the other.

    Copyright 2020 PySimpleGUI.org
"""

def make_win1():
    layout = [[sg.Text('Window 1')],
              [sg.Text('Enter something to output to Window 2')],
              [sg.Input(key='-IN-', enable_events=True)],
              [sg.Text(size=(25,1), key='-OUTPUT-')],
              [sg.Button('Reopen')],
              [sg.Button('Exit')]]
    return sg.Window('Window Title', layout, finalize=True)


def make_win2():
    layout = [[sg.Text('Window 2')],
              [sg.Text('Enter something to output to Window 1')],
              [sg.Input(key='-IN-', enable_events=True)],
              [sg.Text(size=(25,1), key='-OUTPUT-')],
              [sg.Button('Exit')]]
    return sg.Window('Window Title', layout, finalize=True)

def main():
    window1, window2 = make_win1(), make_win2()

    window2.move(window1.current_location()[0], window1.current_location()[1]+220)

    while True:             # Event Loop
        window, event, values = sg.read_all_windows()

        if window == sg.WIN_CLOSED:     # if all windows were closed
            break
        if event == sg.WIN_CLOSED or event == 'Exit':
            window.close()
            if window == window2:       # if closing win 2, mark as closed
                window2 = None
            elif window == window1:     # if closing win 1, mark as closed
                window1 = None
        elif event == 'Reopen':
            if not window2:
                window2 = make_win2()
                window2.move(window1.current_location()[0], window1.current_location()[1] + 220)
        elif event == '-IN-':
            output_window = window2 if window == window1 else window1
            if output_window:           # if a valid window, then output to it
                output_window['-OUTPUT-'].update(values['-IN-'])
            else:
                window['-OUTPUT-'].update('Other window is closed')


if __name__ == '__main__':
    main()
```

Recipe - All Elements (The Everything Bagel... 2022-Style)

As of June 2022, this window represents all 33 of the PySimpleGUI elements in a single window. It takes about 70 lines of code to show all of the elements in this one window.

The code isn't as practical as what you'll find in the Demo Programs. This use of the elements is shortened and doesn't show how each is used in a typical way.


The PySimpleGUI Element List
File Edit
PySimpleGUI Elements - Use Combo to Change Themes
Use Custom Titlebar & Menubar
Text ····· Text
Input ·····
Multiline ·····
Output ·····
Combo ····· DarkBlue3
OptionMenu ·····
Checkbox ····· Checkbox
Radio ····· Radio
Spin ····· Spin
Button ····· Button
ButtonMenu ····· ButtonMenu
Slider ····· 0
Listbox ····· Listbox Listbox 2
Image ·····
Graph ·····
Canvas ·····
ProgressBar ·····
Table ····· Col 1 Col 2 Col 3 1 2 3 4 5 6
Tree ····· Heading Tree Item 1 1234 B
Horizontal Separator ·
Vertical Separator ···
Frame ····· Frame
Column ·····
Tab, TabGroup ····· Tab1 Tab2
Pane ····· Pane 1 Pane 2
Push ····· Pushed over
VPush ·····
Sizer ·····
StatusBar ····· StatusBar
Sizegrip ·····

```
import PySimpleGUI as sg

"""
    Demo - Element List

    All elements shown in 1 window as simply as possible.

    Copyright 2022 PySimpleGUI
"""


use_custom_titlebar = True if sg.running_trinket() else False

def make_window(theme=None):

    NAME_SIZE = 23

    def name(name):
        dots = NAME_SIZE-len(name)-2
        return sg.Text(name + ' ' + '•'*dots, size=(NAME_SIZE,1), justification='r',pad=(0,0), fon

    sg.theme(theme)

    # NOTE that we're using our own LOCAL Menu element
    if use_custom_titlebar:
        Menu = sg.MenubarCustom
    else:
        Menu = sg.Menu

    treedata = sg.TreeData()

    treedata.Insert("", '_A_', 'Tree Item 1', [1234], )
    treedata.Insert("", '_B_', 'B', [])
    treedata.Insert("_A_", '_A1_', 'Sub Item 1', ['can', 'be', 'anything'], )

    layout_l = [
                [name('Text'), sg.Text('Text')],
                [name('Input'), sg.Input(s=15)],
                [name('Multiline'), sg.Multiline(s=(15,2))],
                [name('Output'), sg.Output(s=(15,2))],
                [name('Combo'), sg.Combo(sg.theme_list(), default_value=sg.theme(), s=(15,22), ena
                [name('OptionMenu'), sg.OptionMenu(['OptionMenu',],s=(15,2))],
                [name('Checkbox'), sg.Checkbox('Checkbox')],
                [name('Radio'), sg.Radio('Radio', 1)],
                [name('Spin'), sg.Spin(['Spin',], s=(15,2))],
                [name('Button'), sg.Button('Button')],
                [name('ButtonMenu'), sg.ButtonMenu('ButtonMenu', sg.MENU_RIGHT_CLICK_EDITME_EXIT)]
                [name('Slider'), sg.Slider((0,10), orientation='h', s=(10,15))],
                [name('Listbox'), sg.Listbox(['Listbox', 'Listbox 2'], no_scrollbar=True,  s=(15,2
                [name('Image'), sg.Image(sg.EMOJI_BASE64_HAPPY_THUMBS_UP)],
                [name('Graph'), sg.Graph((125, 50), (0,0), (125,50), k='-GRAPH-')]  ]

    layout_r  = [[name('Canvas'), sg.Canvas(background_color=sg.theme_button_color()[1], size=(125
                [name('ProgressBar'), sg.ProgressBar(100, orientation='h', s=(10,20), k='-PBAR-')]
                [name('Table'), sg.Table([[1,2,3], [4,5,6]], ['Col 1','Col 2','Col 3'], num_rows=2
                [name('Tree'), sg.Tree(treedata, ['Heading',], num_rows=3)],
                [name('Horizontal Separator'), sg.HSep()],
                [name('Vertical Separator'), sg.VSep()],
                [name('Frame'), sg.Frame('Frame', [[sg.T(s=15)]])],
                [name('Column'), sg.Column([[sg.T(s=15)]])],
                [name('Tab, TabGroup'), sg.TabGroup([[sg.Tab('Tab1',[[sg.T(s=(15,2))]]), sg.Tab('T
                [name('Pane'), sg.Pane([sg.Col([[sg.T('Pane 1')]]), sg.Col([[sg.T('Pane 2')]])])],
                [name('Push'), sg.Push(), sg.T('Pushed over')],
                [name('VPush'), sg.VPush()],
                [name('Sizer'), sg.Sizer(1,1)],
                [name('StatusBar'), sg.StatusBar('StatusBar')],
                [name('Sizegrip'), sg.Sizegrip()]  ]

    # Note - LOCAL Menu element is used (see about for how that's defined)
    layout = [[Menu([['File', ['Exit']], ['Edit', ['Edit Me', ]]],  k='-CUST MENUBAR-',p=0)],
              [sg.T('PySimpleGUI Elements - Use Combo to Change Themes', font='_ 14', justificatio
              [sg.Checkbox('Use Custom Titlebar & Menubar', use_custom_titlebar, enable_events=Tru
              [sg.Col(layout_l, p=0), sg.Col(layout_r, p=0)]]

    window = sg.Window('The PySimpleGUI Element List', layout, finalize=True, right_click_menu=sg.

    window['-PBAR-'].update(30)                                                 # Show 30% com
    window['-GRAPH-'].draw_image(data=sg.EMOJI_BASE64_HAPPY_JOY, location=(0,50))   # Draw somethi

    return window


window = make_window()
```

```
while True:
    event, values = window.read()
    # sg.Print(event, values)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == 'Edit Me':
        sg.execute_editor(__file__)
    if values['-COMBO-'] != sg.theme():
        sg.theme(values['-COMBO-'])
        window.close()
        window = make_window()
    if event == '-USE CUSTOM TITLEBAR-':
        use_custom_titlebar = values['-USE CUSTOM TITLEBAR-']
        sg.set_options(use_custom_titlebar=use_custom_titlebar)
        window.close()
        window = make_window()
    elif event == 'Version':
        sg.popup_scrolled(sg.get_versions(), __file__, keep_on_top=True, non_blocking=True)
window.close()
```

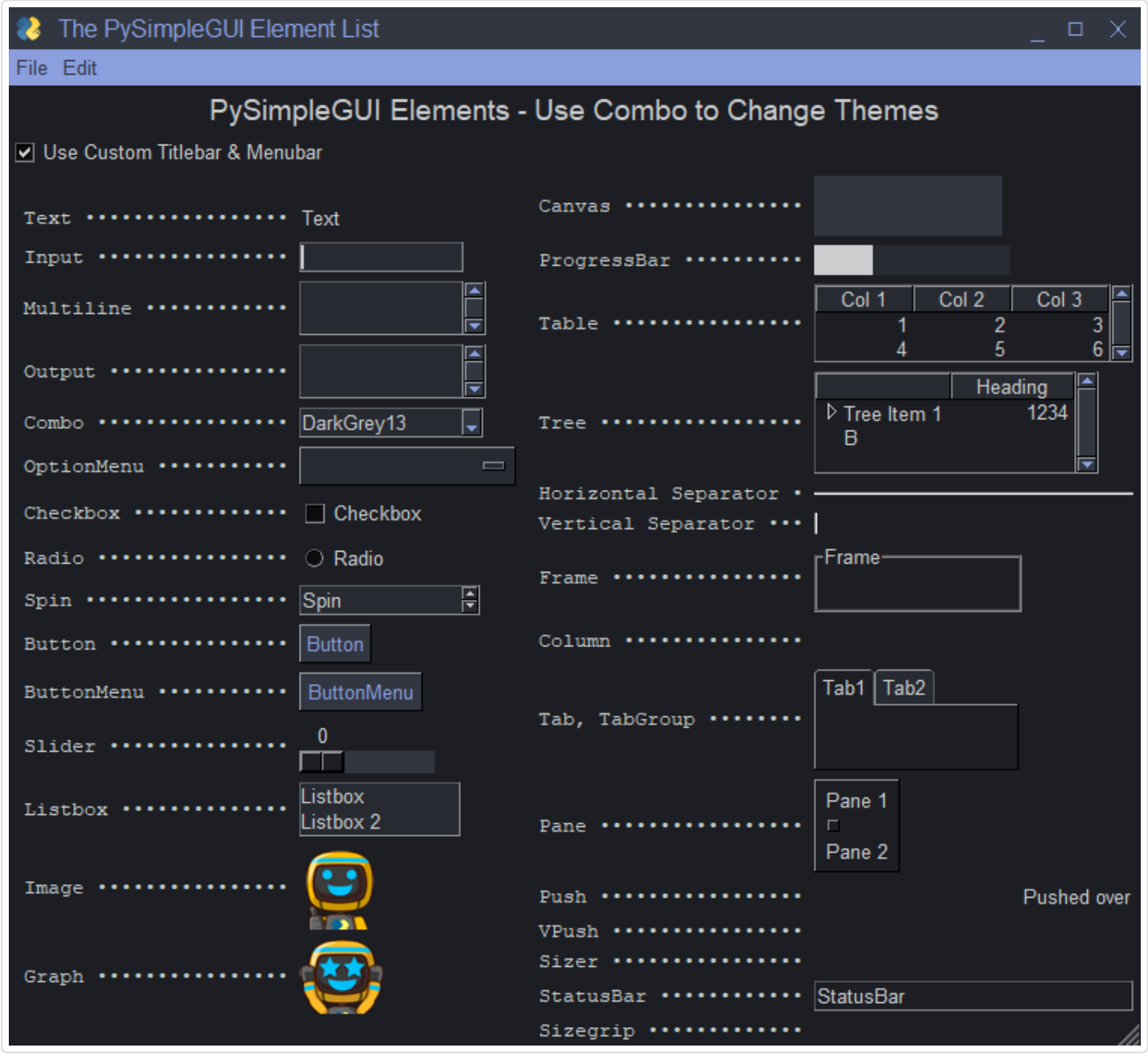
Custom Titlebar and Menubar

The checkbox at the top of the left column can be checked in order to display the window with the custom titlebar and custom menubar

Can use to preview Themes

One practical use of this Demo Program is that you can preview how the elements look when using a specific theme by choosing the theme using the Combo element in the window.

For example, choosing the DarkGrey13 theme, with the custom titlebar checked, the window is replaced with one using that theme. Very nice!



Recipe - Exception handling of a console-less application

Some applications, such as those that run in the system tray or are "Desktop Widgets" are meant to run entirely without a console. On Windows, they're often named with a .pyw extension. This is challenging in a couple of ways.

1. These programs run for days rather than moments
2. They have no console to output to

Debugging an intermittent problem is difficult when using plain Python for an application like these. Normally you would need to use a debugger so that the crash is caught when it happens or use a logging module that writes to disk.

With PySimpleGUI there are a number of simple ways to handle these types of errors, assuming that PySimpleGUI itself has remained functional.

Numerous `popup` calls are available to output information to a popup window. If you want to output a lot of information that you can also copy and paste from, then a `popup_scrolled` call is a good choice. However, there's an ever better technique.

The Debug Print Window

The Debug Print Window is One of the easiest ways to get the information to help you debug the problem. To output to this window, call the function `sg.Print` in the same way you a normal `print`. It accepts similar parameters including `sep` and `end` as well as some PySimpleGUI specific ones that enable printing in color.

One advantage that the Debug Print has over popups is that you can output the information using multiple calls. Each time you call `sg.Print`, the information will be added to output that's already in the Debug Print Window. You will be able to copy and paste from this window to another application should there be a log or some other information that you can to save. Maybe you want to loop through a list and print certain items as part of the information to display. Using `sg.Print` will enable you to easily do this.

Use the `wait` / `blocking` parameter!

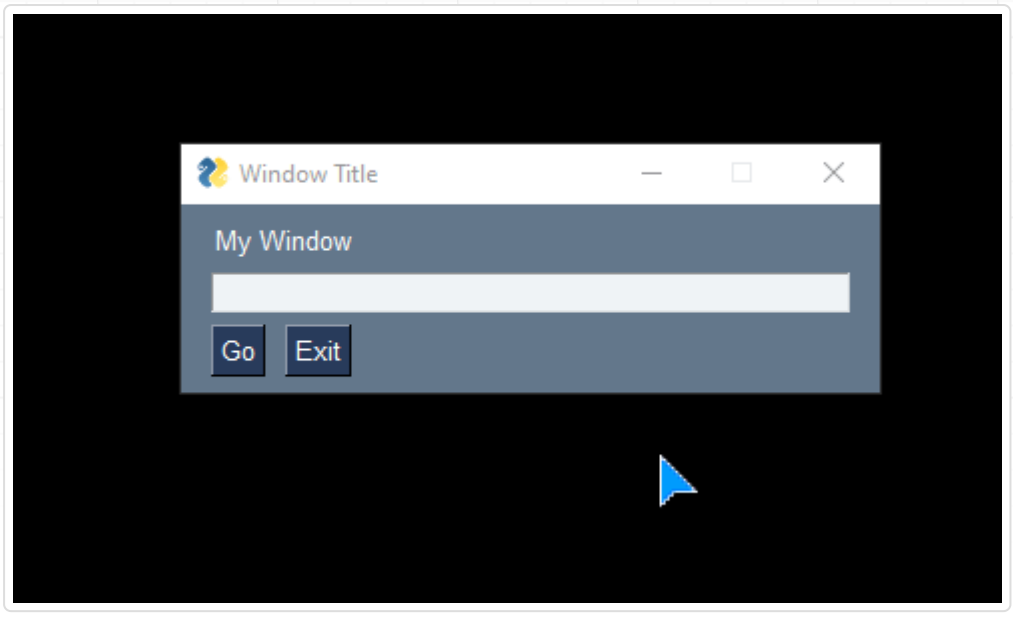
Be sure and set the `wait` parameter to `True` in the last call you make to `sg.Print`. The reason for this is that if you do not, the call will output to the window and then immediately return and your program will likely continue on and exit. Adding `wait=True` will output to the Debug Window and then wait for you to click the `Continue` button in the window before the call will return.

Example Program With Exception

This example will crash when you click the "Go" button.

```
import PySimpleGUI as sg

def main():
    layout = [  [sg.Text('My Window')],
                [sg.Input(key='-IN-')],
                [sg.Button('Go'), sg.Button('Exit')]  ]

    window = sg.Window('Window Title', layout)

    while True:             # Event Loop
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Exit':
            break
        if event == 'Go':
            bad_call()
    window.close()

if __name__ == '__main__':
    main()
```

If this program was launched by double clicking a .pyw file or some other technique that doesn't involve a debugger or command line, then what you would see is this:



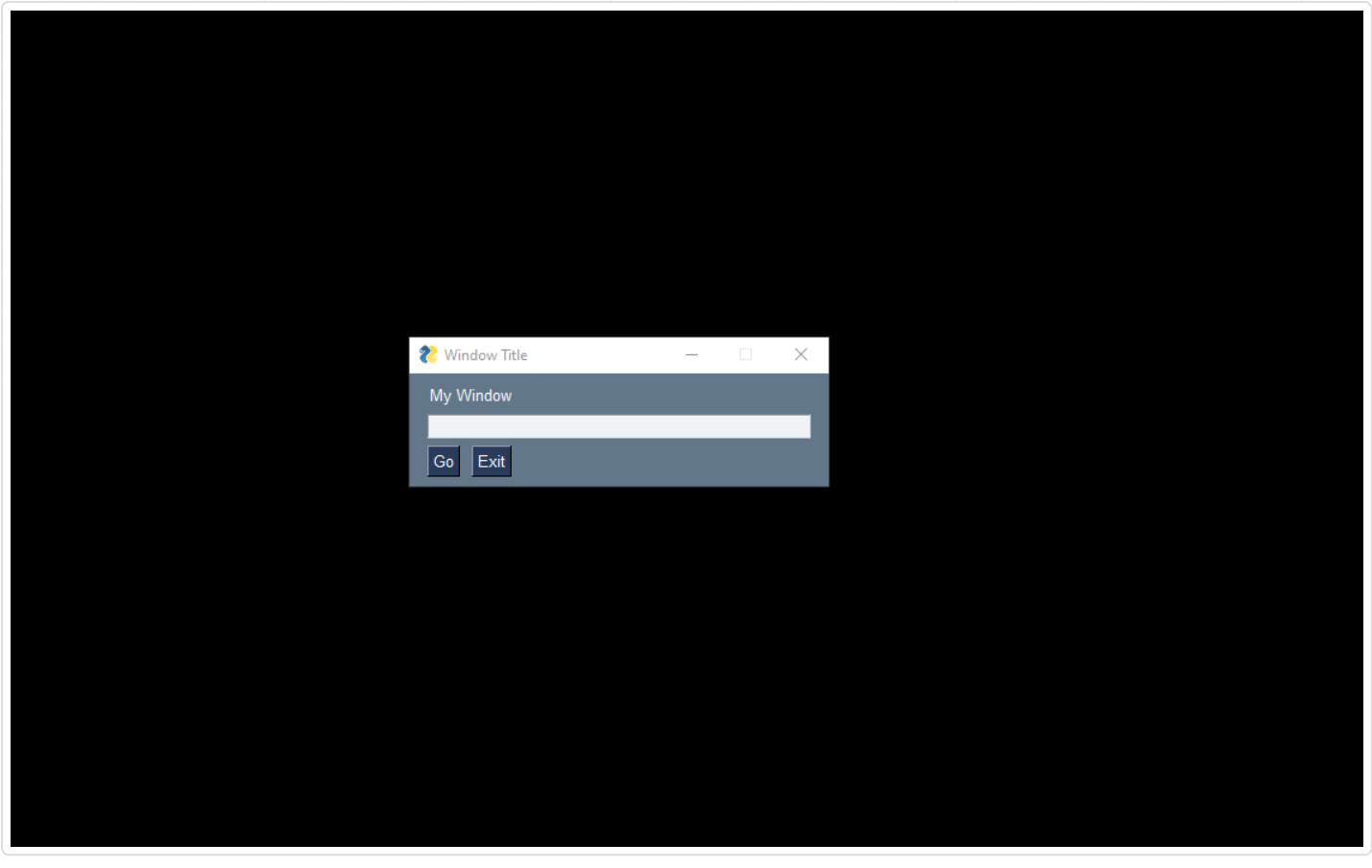
Your window would disappear and you would have nothing to help you understand why. This is why PySimpleGUI uses a popup window for errors that are encountered internally instead of raising an exception.

Example With Exception Handling Added

Here is the same program, but we've added a `try` block around the entire event loop. A total of 3 line of code were added.
1. The `try` statement 2. The `except` statement 3. The `sg.Print` call to output to the Debug Window

```
import PySimpleGUI as sg

def main():
    layout = [  [sg.Text('My Window')],
                [sg.Input(key='-IN-')],
                [sg.Button('Go'), sg.Button('Exit')]  ]

    window = sg.Window('Window Title', layout)

    try:
        while True:             # Event Loop
            event, values = window.read()
            if event == sg.WIN_CLOSED or event == 'Exit':
                break
            if event == 'Go':
                bad_call()
    except Exception as e:
        sg.Print('Exception in my event loop for the program:', sg.__file__, e, keep_on_top=True,

    window.close()

if __name__ == '__main__':
    main()
```

This time the experience enables you to understand why and where your program crashed. It's clearly there's a problem with the call to `bad_call`



Adding Traceback and Editing Capabilities

With the Debug Print, you're able to output information about the exception which certainly helps. Because PySimpleGUI was written for you, a Python developer, there's a popup call that you can add to make your life even *easier*!

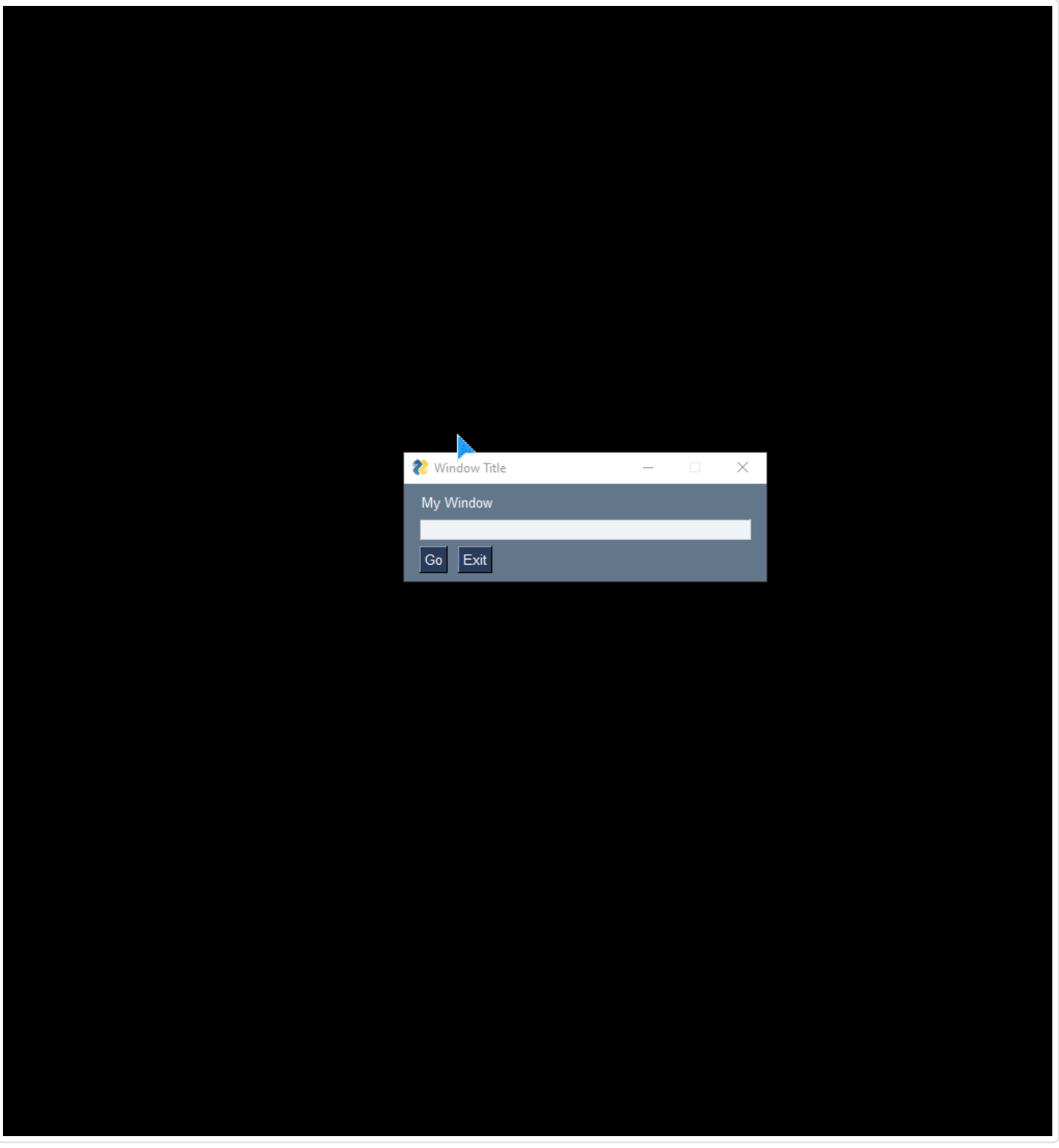
We'll change the exception handling portion this time to be these 2 lines of code:

```
        sg.Print('Exception in my event loop for the program:', sg.__file__, e, keep_on_top=True)
        sg.popup_error_with_traceback('Problem in my event loop!', e)
```

We're still print to the Debug Window, but we've removed the `wait` parameter because the `popup` call immediately after it will stop the program from exiting. The popup we've added is `popup_error_with_traceback`. This call will show you more detailed information about where the problem happened, and more conveniently, enable you to open your editor to the line of code that caused the problem. You will need to add your editor's information in the PySimpleGUI globals settings for this feature to work, so be sure and fill in this information.

Now our experience changes considerably. We get the Debug Window as before, but we also get a popup that has a "Take me to error" button that will open your editor to the line of code where you called the popup. The exception information is included in the popup because we added it when calling the popup.

This popup saves you the time of locating where on your computer the .py or .pyw file is located. Just click the button and your editor will be launched with the correct filename and you'll be taken to the line of code. You also get, for free, an attempt at humor with an unhappy looking emoji.



Asynchronous Window With Periodic Update

Sync Versus Async Mode

It's possible, and even easy, to run your PySimpleGUI program in an "asynchronous" way.

What does that even mean?

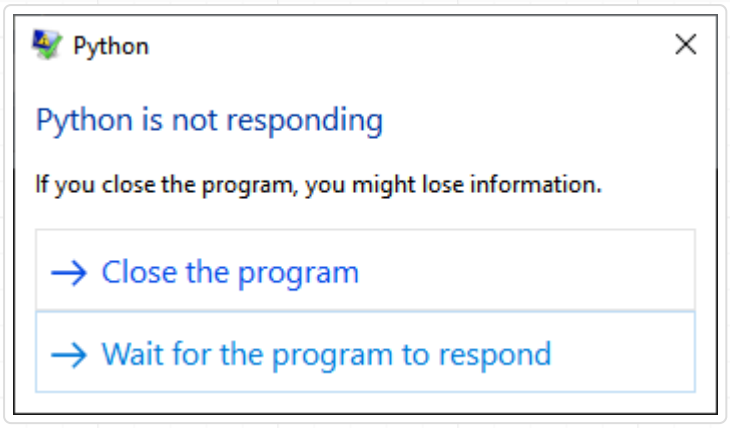
There are 2 modes sync and async. When running normally (synchronous), calls are made into the GUI ***stay*** in the GUI until something happens. You call `window.read()` and wait for a button or some event that causes the `read` to return.

With async calls, you wait for an event for a certain amount of time and then you return after that amount of time if there's no event. You don't wait forever for a new event.

When running asynchronously, you are giving the illusion that multiple things are happening at the same time when in fact they are interwoven.

It means your program doesn't "block" or stop running while the user is interacting with the window. Your program continues to run and does things while the user is fiddling around.

The critical part of these async windows is to ensure that you are calling either `read` or `refresh` often enough. Just because your code is running doesn't mean you can ignore the GUI. We've all experienced what happens when a GUI program "locks up". We're shown this lovely window.



This happens when the GUI subsystem isn't given an opportunity to run for a long time. Adding a sleep to your event loop will cause one of these to pop up pretty quickly.

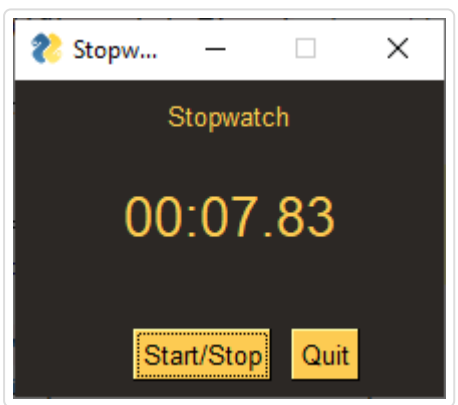
We can "cheat" a little though. Rather than being stuck inside the GUI code, we get control back, do a little bit of work, and then jump back into the GUI code. If this is done quickly enough, you don't get the ugly little "not responding" window.

Async Uses - Polling

Use this design pattern for projects that need to poll or output something on a regular basis. In this case, we're indicating we want a `timeout=10` on our `window.read` call. This will cause the `Read` call to return a "timeout key" as the event every 10 milliseconds has passed without some GUI thing happening first (like the user clicking a button). The timeout key is `PySimpleGUI.TIMEOUT_KEY` usually written as `sg.TIMEOUT_KEY` in normal PySimpleGUI code.

Use caution when using windows with a timeout. You should **rarely** need to use a `timeout=0`. A zero value is a truly non-blocking call, so try not to abuse this design pattern. You shouldn't use a timeout of zero unless you're a realtime application and you know what you're doing. A zero value will consume 100% of the CPU core your code is running on. Abuse it an bad things ***will*** happen.

A note about timers... this is not a good design for a stopwatch as it can very easily drift. This would never pass for a good solution in a bit of commercial code. For better accuracy always get the actual time from a reputable source, like the operating system. Use that as what you use to measure and display the time.

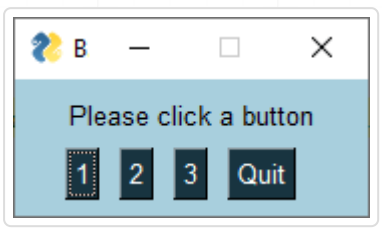



```
import PySimpleGUI as sg

sg.theme('DarkBrown1')

layout = [  [sg.Text('Stopwatch', size=(20, 2), justification='center')],
            [sg.Text(size=(10, 2), font=('Helvetica', 20), justification='center', key='-OUTPUT-')
            [sg.T(' ' * 5), sg.Button('Start/Stop', focus=True), sg.Quit()]]

window = sg.Window('Stopwatch Timer', layout)

timer_running, counter = True, 0

while True:                                 # Event Loop
    event, values = window.read(timeout=10) # Please try and use as high of a timeout value as you
    if event in (sg.WIN_CLOSED, 'Quit'):            # if user closed the window using X or clicke
        break
    elif event == 'Start/Stop':
        timer_running = not timer_running
    if timer_running:
        window['-OUTPUT-'].update('{:02d}:{:02d}.{:02d}'.format((counter // 100) // 60, (counter /
        counter += 1
window.close()
```

The `focus` parameter for the Button causes the window to start with that button having focus. This will allow you to press the return key or spacebar to control the button.

Recipe - Callback Function Simulation

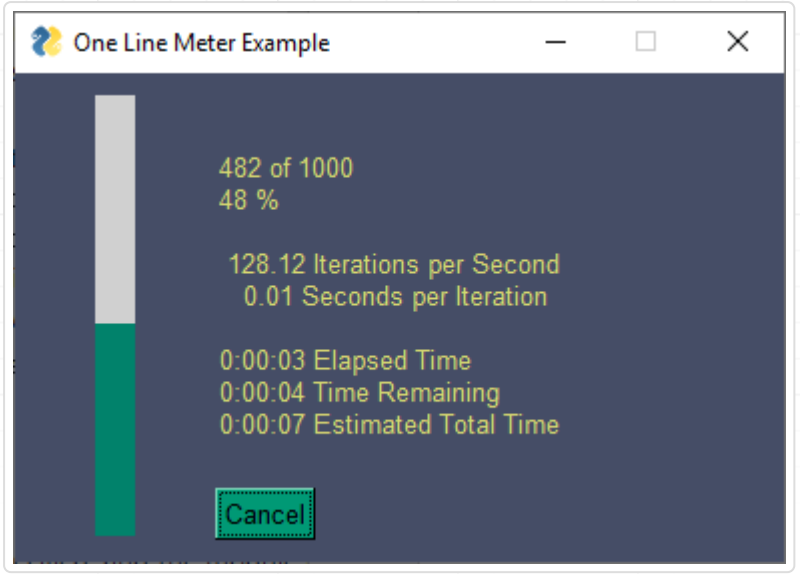
The architecture of some programs works better with button callbacks instead of handling in-line. While button callbacks are part of the PySimpleGUI implementation, they are not directly exposed to the caller. The way to get the same result as callbacks is to simulate them with a recipe like this one.




```
import PySimpleGUI as sg

sg.theme('Light Blue 3')
# This design pattern simulates button callbacks
# This implementation uses a simple "Dispatch Dictionary" to store events and functions

# The callback functions
def button1():
    print('Button 1 callback')

def button2():
    print('Button 2 callback')

# Lookup dictionary that maps button to function to call
dispatch_dictionary = {'1':button1, '2':button2}

# Layout the design of the GUI
layout = [[sg.Text('Please click a button', auto_size_text=True)],
          [sg.Button('1'), sg.Button('2'), sg.Button('3'), sg.Quit()]]

# Show the Window to the user
window = sg.Window('Button callback example', layout)

# Event loop. Read buttons, make callbacks
while True:
    # Read the Window
    event, value = window.read()
    if event in ('Quit', sg.WIN_CLOSED):
        break
    # Lookup event in function dictionary
    if event in dispatch_dictionary:
        func_to_call = dispatch_dictionary[event]   # get function from dispatch dictionary
        func_to_call()
    else:
        print('Event {} not in dispatch dictionary'.format(event))

window.close()

    # All done!
sg.popup_ok('Done')
```

Recipe - one_line_progress_meter

This recipe shows just how easy it is to add a progress meter to your code.




```
import PySimpleGUI as sg

sg.theme('Dark Blue 8')

for i in range(1000):   # this is your "work loop" that you want to monitor
    sg.one_line_progress_meter('One Line Meter Example', i + 1, 1000)
```

Unlike other progress meter Python packages, PySimpleGUI's one-line-progress-meter is 1 line of code, not 2. Historicly you would setup the meter outside your work loop and then update that meter inside of your loop. With PySimpleGUI you do not need to setup the meter outside the loop. You only need to add the line of code to update the meter insdie of your loop.

Cancelling - User Generated

If you want to enable the user to break out of your loop, then **you** will need to take action. `one_line_progress_meter` returns `False` if the cancel button is clicked or the window is somehow closed. The way you can turn the user's click on the Cancel button into cancelling the loop is by checking the return value and then breaking from your loop.

```
import PySimpleGUI as sg

sg.theme('Dark Blue 8')

for i in range(1000):   # this is your "work loop" that you want to monitor
    if not sg.one_line_progress_meter('One Line Meter Example', i + 1, 1000):
        break
```

Cancelling - Program Generated

You've seen how the user can cancel the progress meter, now let's look at how your program can cancel the progress meter. Maybe you got an error 1/2 way through the processing and you want to stop the meter. To cancel the `one_line_progress_meter` ... I bet you can't guess what function you would call....

```
import PySimpleGUI as sg

sg.theme('Dark Blue 8')

for i in range(1000):   # this is your "work loop" that you want to monitor
    sg.one_line_progress_meter('One Line Meter Example', i + 1, 1000)
    # after 500 iterations, cancel the meter
    if i == 500:
        sg.one_line_progress_meter_cancel()
        sg.popup('Cancelled!')
        break
```

Other Parameters

Care is neeeded when it comes to the parameters after the first 3 parms. You can add any number of variable arguments to be shown in the progress meter window.

The first 3 parms are required. Then there are any number of parms you can add to your progress meter window. After those then there are other parameters after the variable numbered *args. Here's the definition (see the Call Reference Documentaion for the most up to date version)

```
def one_line_progress_meter(title, current_value, max_value, *args, key='OK for 1 meter', orientat
```

If you wanted your meter to be horizontal instead of vertical and include some additional information, then your call may look like this:

```
    sg.one_line_progress_meter('One Line Meter Example', i + 1, 1000, 'This is my custom message',
```

Keys

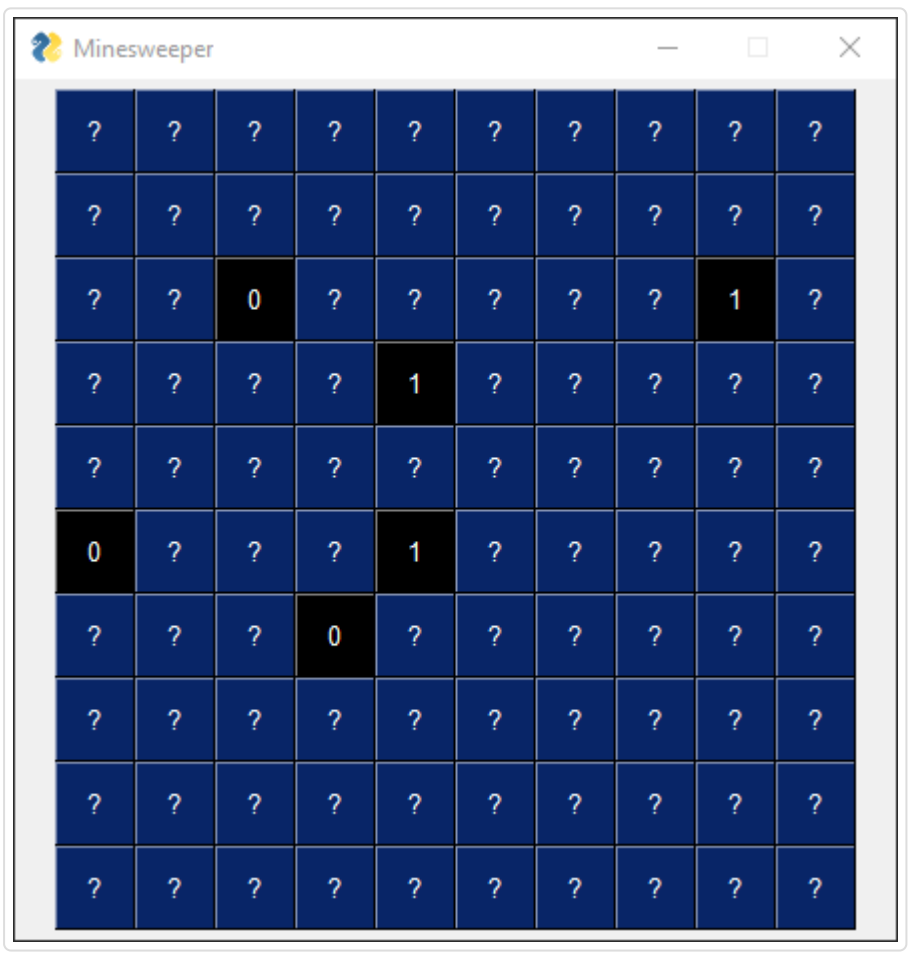
If you have only 1 one_line_progress_meter running at a time, then you don't need to set a key. The default keyt is fine. If you want multiple windows running simultaneously, then you will need to set keys for each window. You will need to use the same keys for cancelling the meter early.

Here's the example with the cancel using a custom key:

```
import PySimpleGUI as sg

sg.theme('Dark Blue 8')

for i in range(1000):   # this is your "work loop" that you want to monitor
    sg.one_line_progress_meter('One Line Meter Example', i + 1, 1000, key=1)
    # after 500 iterations, cancel the meter
    if i == 500:
        sg.one_line_progress_meter_cancel(key=1)
        sg.popup('Cancelled!')
        break
```

Recipe - Minesweeper-style Grid of Buttons

There are a number of applications built using a GUI that involve a grid of buttons. The games Minesweeper and Battleship can both be thought of as a grid of buttons.



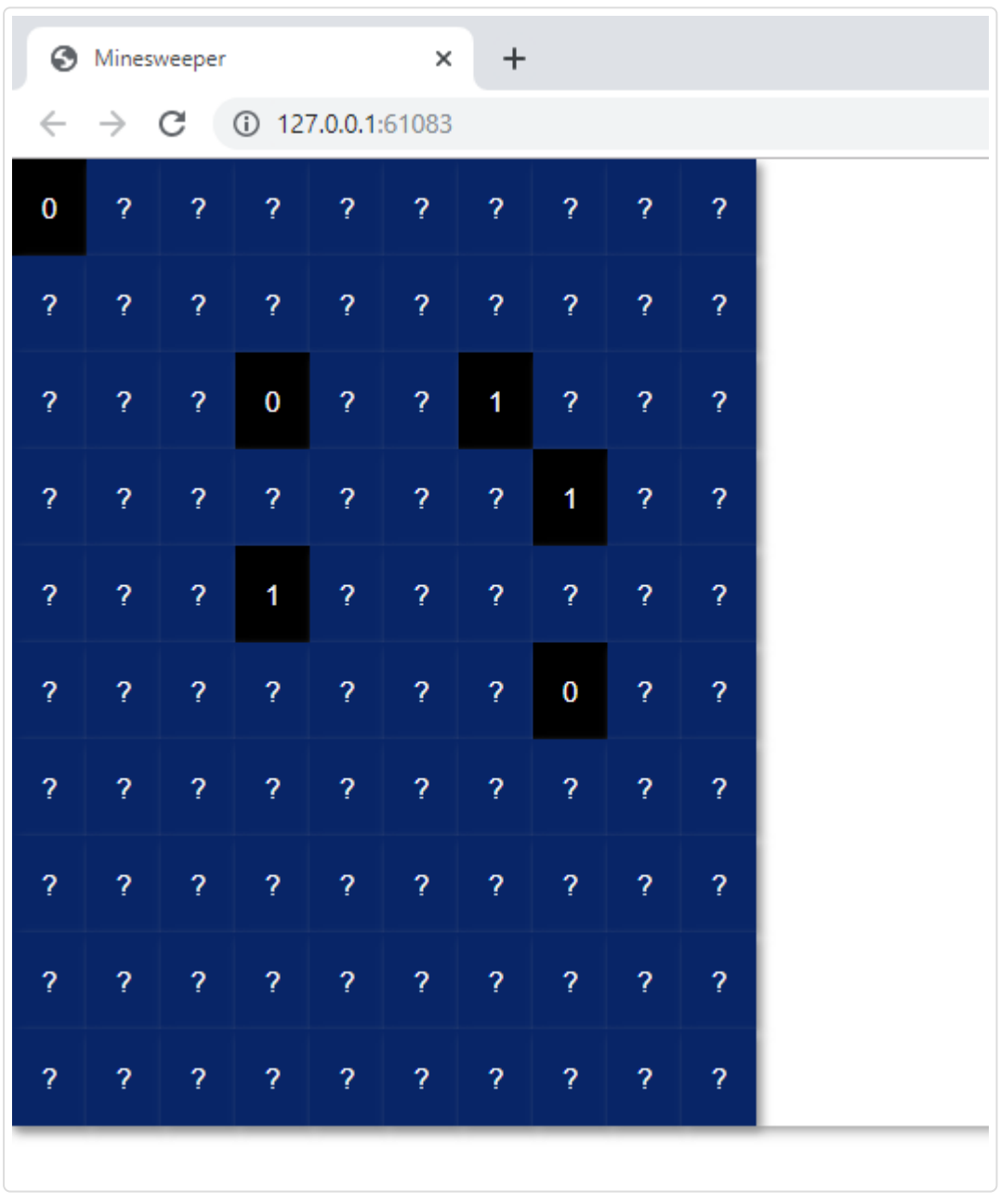
Here is the code for the above window

```
import PySimpleGUIWeb as sg
from random import randint

MAX_ROWS = MAX_COL = 10
board = [[randint(0,1) for j in range(MAX_COL)] for i in range(MAX_ROWS)]

layout =  [[sg.Button('?', size=(4, 2), key=(i,j), pad=(0,0)) for j in range(MAX_COL)] for i in ra

window = sg.Window('Minesweeper', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Exit'):
        break
    # window[(row, col)].update('New text')   # To change a button's text, use this pattern
    # For this example, change the text of the button to the board's value and turn color black
    window[event].update(board[event[0]][event[1]], button_color=('white','black'))
window.close()
```

The **most important** thing for you to learn from this recipe is that keys and events can be **any type**, not just strings.

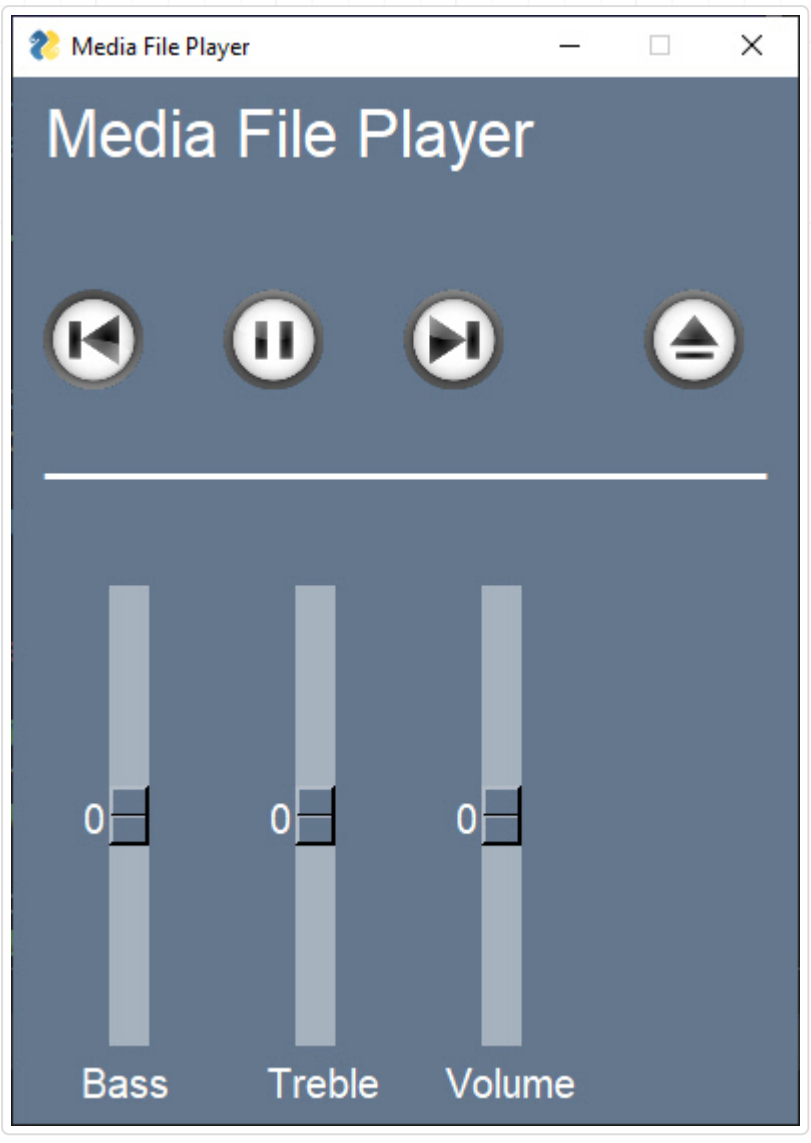
Thinking about this grid of buttons, doesn't it make the most sense for you to get row, column information when a button is pressed. Well, that's exactly what setting your keys for these buttons to be tuples does for you. It gives you the abilty to read events and finding the button row and column, and it makes updating text or color of buttons using a row, column designation.

This program also runs on PySimpleGUIWeb really well. Change the import to PySimpleGUIWeb and you'll see this in your web browser (assuming you've installed PySimpleGUIWeb)



Recipe - Button Graphics (Media Player)

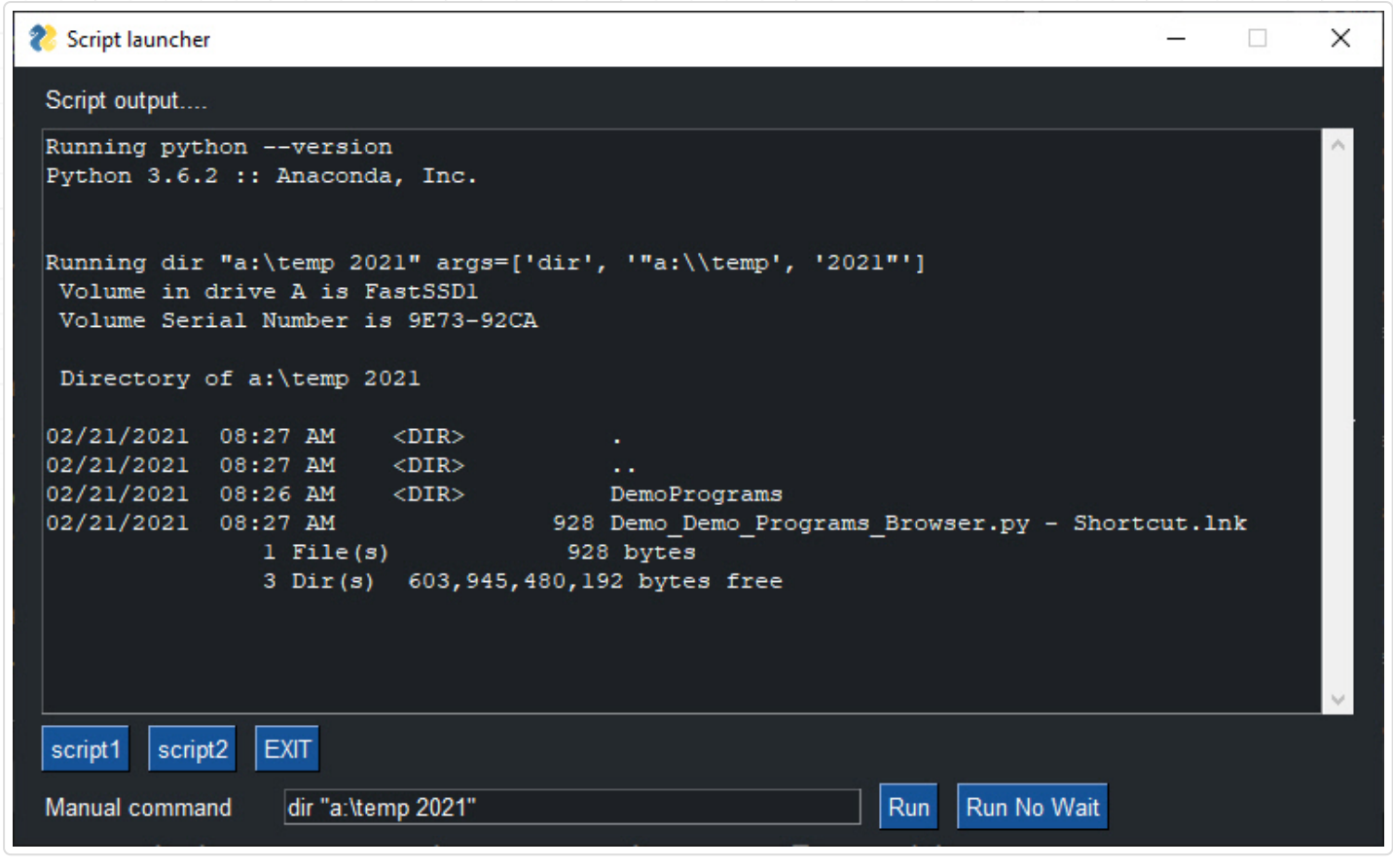
Buttons can have PNG of GIF images on them. This Media Player recipe requires 4 images in order to function correctly. The background is set to the same color as the button background so that they blend together.



```
#!/usr/bin/env python
import PySimpleGUI as sg

#
# An Async Demonstration of a media player
# Uses button images for a super snazzy look
# See how it looks here:
# https://user-images.githubusercontent.com/13696193/43159403-45c9726e-8f50-11e8-9da0-0d272e20c579
#
def MediaPlayerGUI():
    # Set the backgrounds the same as the background on the buttons
    # Images are located in a subfolder in the Demo Media Player.py folder
    image_pause = './ButtonGraphics/Pause.png'
    image_restart = './ButtonGraphics/Restart.png'
    image_next = './ButtonGraphics/Next.png'
    image_exit = './ButtonGraphics/Exit.png'

    # Use the theme APIs to set the buttons to blend with background
    sg.theme_button_color((sg.theme_background_color(), sg.theme_background_color()))
    sg.theme_border_width(0)        # make all element flat

    # define layout of the rows
    layout= [[sg.Text('Media File Player',size=(17,1), font=("Helvetica", 25))],
             [sg.Text(size=(15, 2), font=("Helvetica", 14), key='-OUTPUT-')],
             [sg.Button(image_filename=image_restart, image_size=(50, 50), image_subsample=2,  key
              sg.Text(' ' * 2),
              sg.Button(image_filename=image_pause, image_size=(50, 50), image_subsample=2,  key='
              sg.Text(' ' * 2),
              sg.Button(image_filename=image_next, image_size=(50, 50), image_subsample=2,  key='-
              sg.Text(' ' * 2),
              sg.Text(' ' * 2), sg.Button(image_filename=image_exit, image_size=(50, 50), image_su
             [sg.Text('_'*20)],
             [sg.Text(' '*30)],
             [
                 sg.Slider(range=(-10, 10), default_value=0, size=(10, 20), orientation='vertical'
                 sg.Text(' ' * 2),
                 sg.Slider(range=(-10, 10), default_value=0, size=(10, 20), orientation='vertical'
                 sg.Text(' ' * 2),
                 sg.Slider(range=(-10, 10), default_value=0, size=(10, 20), orientation='vertical'
             [sg.Text('   Bass', font=("Helvetica", 15), size=(9, 1)),
              sg.Text('Treble', font=("Helvetica", 15), size=(7, 1)),
              sg.Text('Volume', font=("Helvetica", 15), size=(7, 1))]
             ]

    # Open a form, note that context manager can't be used generally speaking for async forms
    window = sg.Window('Media File Player', layout, default_element_size=(20, 1), font=("Helvetica
    # Our event loop
    while True:
        event, values = window.read(timeout=100)        # Poll every 100 ms
        if event == 'Exit' or event == sg.WIN_CLOSED:
            break
        # If a button was pressed, display it on the GUI by updating the text element
        if event != sg.TIMEOUT_KEY:
            window['-OUTPUT-'].update(event)

MediaPlayerGUI()
```

Recipe - Script Launcher - Exec APIs

This program will run commands and display the output in an Output Element. There are numerous Demo Programs that show how to lauch subprocesses manually rather than using the newer Exec APIs.




```
import PySimpleGUI as sg

sg.theme('DarkGrey14')

layout = [
    [sg.Text('Script output....', size=(40, 1))],
    [sg.Output(size=(88, 20), font='Courier 10')],
    [sg.Button('script1'), sg.Button('script2'), sg.Button('EXIT')],
    [sg.Text('Manual command', size=(15, 1)), sg.Input(focus=True, key='-IN-'), sg.Button('Run', b
]

window = sg.Window('Script launcher', layout)

# ---===--- Loop taking in user input and using it to call scripts --- #

while True:
    event, values = window.read()
    if event == 'EXIT'  or event == sg.WIN_CLOSED:
        break # exit button clicked
    if event == 'script1':
        sp = sg.execute_command_subprocess('pip', 'list', wait=True)
        print(sg.execute_get_results(sp)[0])
    elif event == 'script2':
        print(f'Running python --version')
        # For this one we need to wait for the subprocess to complete to get the results
        sp = sg.execute_command_subprocess('python', '--version', wait=True)
        print(sg.execute_get_results(sp)[0])
    elif event == 'Run':
        args = values['-IN-'].split(' ')
        print(f'Running {values["-IN-"]} args={args}')
        sp = sg.execute_command_subprocess(args[0], *args[1:])
        # This will cause the program to wait for the subprocess to finish
        print(sg.execute_get_results(sp)[0])
    elif event == 'Run No Wait':
        args = values['-IN-'].split(' ')
        print(f'Running {values["-IN-"]} args={args}', 'Results will not be shown')
        sp = sg.execute_command_subprocess(args[0], *args[1:])
```

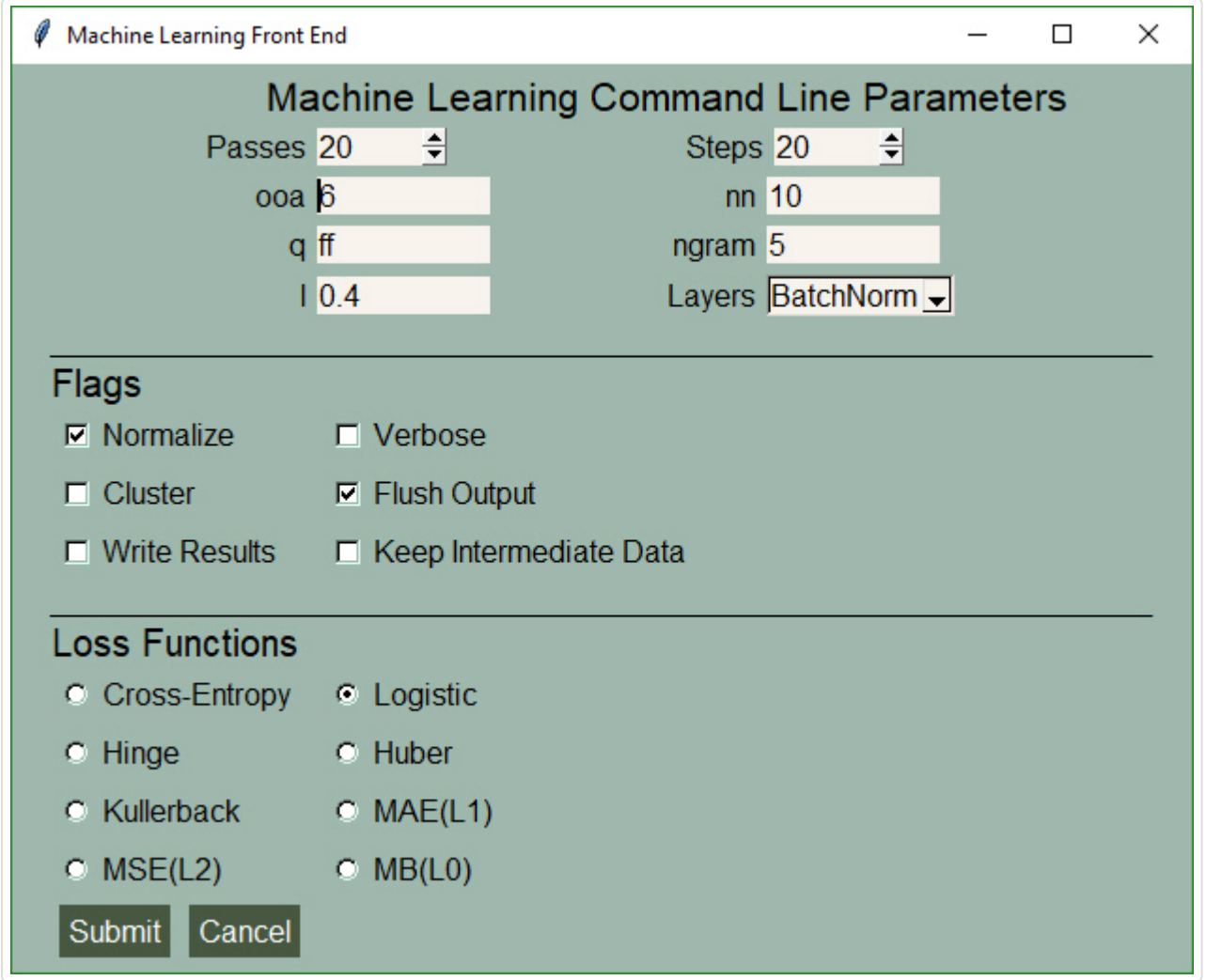
Recipe- Launch a Program With a Button

Very simple script that will launch a program as a subprocess. Great for making a desktop launcher toolbar. In version 4.35.0 of PySimpleGUI the Exec APIs were added. These enable you to launch subprocesses

```
import PySimpleGUI as sg

CHROME = r"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe"


layout = [  [sg.Text('Text area', key='-TEXT-')],
            [sg.Input(key='-URL-')],
            [sg.Button('Chrome'), sg.Button('Exit')]]

window = sg.Window('Window Title', layout)

while True:             # Event Loop
    event, values = window.read()
    print(event, values)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    if event == 'Chrome':
        sg.execute_command_subprocess(CHROME)

window.close()
```

Recipe - Machine Learning GUI

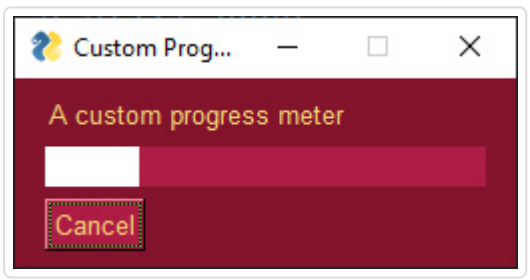
A standard non-blocking GUI with lots of inputs.



```
    import PySimpleGUI as sg

    # Green & tan color scheme
    sg.theme('GreenTan')

    sg.set_options(text_justification='right')

    layout = [[sg.Text('Machine Learning Command Line Parameters', font=('Helvetica', 16))],
              [sg.Text('Passes', size=(15, 1)), sg.Spin(values=[i for i in range(1, 1000)], initia
               sg.Text('Steps', size=(18, 1)), sg.Spin(values=[i for i in range(1, 1000)], initial
              [sg.Text('ooa', size=(15, 1)), sg.In(default_text='6', size=(10, 1)), sg.Text('nn',
               sg.In(default_text='10', size=(10, 1))],
              [sg.Text('q', size=(15, 1)), sg.In(default_text='ff', size=(10, 1)), sg.Text('ngram'
               sg.In(default_text='5', size=(10, 1))],
              [sg.Text('l', size=(15, 1)), sg.In(default_text='0.4', size=(10, 1)), sg.Text('Layer
               sg.Drop(values=('BatchNorm', 'other'), auto_size_text=True)],
              [sg.Text('_'  * 100, size=(65, 1))],
              [sg.Text('Flags', font=('Helvetica', 15), justification='left')],
              [sg.Checkbox('Normalize', size=(12, 1), default=True), sg.Checkbox('Verbose', size=(
              [sg.Checkbox('Cluster', size=(12, 1)), sg.Checkbox('Flush Output', size=(20, 1), def
              [sg.Checkbox('Write Results', size=(12, 1)), sg.Checkbox('Keep Intermediate Data', s
              [sg.Text('_'  * 100, size=(65, 1))],
              [sg.Text('Loss Functions', font=('Helvetica', 15), justification='left')],
              [sg.Radio('Cross-Entropy', 'loss', size=(12, 1)), sg.Radio('Logistic', 'loss', defau
              [sg.Radio('Hinge', 'loss', size=(12, 1)), sg.Radio('Huber', 'loss', size=(12, 1))],
              [sg.Radio('Kullerback', 'loss', size=(12, 1)), sg.Radio('MAE(L1)', 'loss', size=(12,
              [sg.Radio('MSE(L2)', 'loss', size=(12, 1)), sg.Radio('MB(L0)', 'loss', size=(12, 1))
              [sg.Submit(), sg.Cancel()]]

    window = sg.Window('Machine Learning Front End', layout, font=("Helvetica", 12))

    event, values = window.read()
```

Recipe - Custom Progress Meter / Progress Bar

Perhaps you don't want all the statistics that the EasyProgressMeter provides and want to create your own progress bar. Use this recipe to do just that.



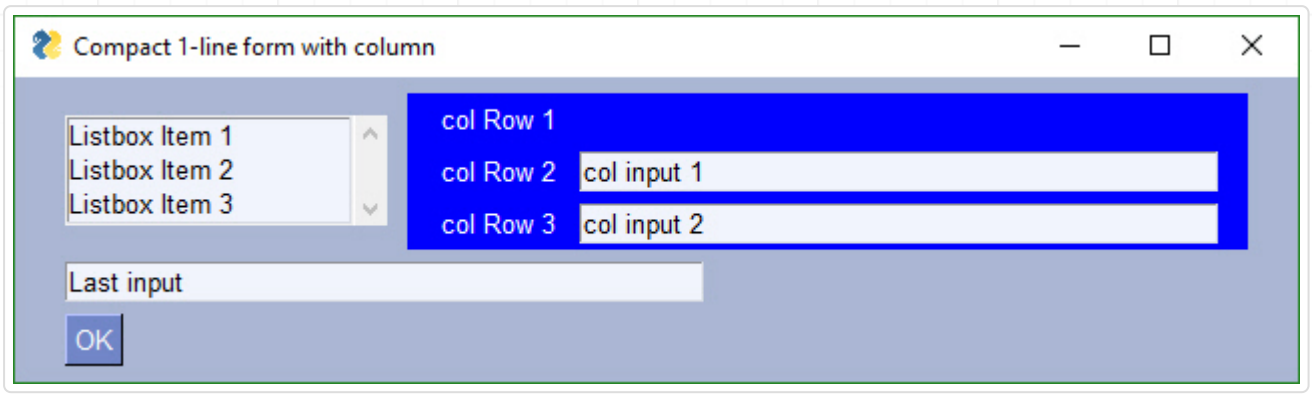

```
import PySimpleGUI as sg

sg.theme('Dark Red')

BAR_MAX = 1000

# layout the Window
layout = [[sg.Text('A custom progress meter')],
          [sg.ProgressBar(BAR_MAX, orientation='h', size=(20,20), key='-PROG-')],
          [sg.Cancel()]]

# create the Window
window = sg.Window('Custom Progress Meter', layout)
# loop that would normally do something useful
for i in range(1000):
    # check to see if the cancel button was clicked and exit loop if clicked
    event, values = window.read(timeout=10)
    if event == 'Cancel' or event == sg.WIN_CLOSED:
        break
        # update bar with loop value +1 so that bar eventually reaches the maximum
    window['-PROG-'].update(i+1)
# done with loop... need to destroy the window as it's still open
window.close()
```

Recipe - Multiple Columns

A Column is required when you have a tall element to the left of smaller elements.

In this example, there is a Listbox on the left that is 3 rows high. To the right of it are 3 single rows of text and input. These 3 rows are in a Column Element.

To make it easier to see the Column in the window, the Column background has been shaded blue. The code is wordier than normal due to the blue shading. Each element in the column needs to have the color set to match blue background.

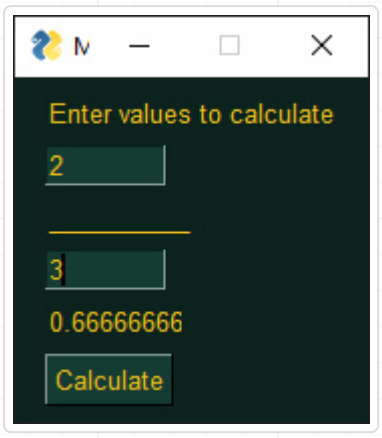



```
import PySimpleGUI as sg

# Demo of how columns work
# GUI has on row 1 a vertical slider followed by a COLUMN with 7 rows
# Prior to the Column element, this layout was not possible
# Columns layouts look identical to GUI layouts, they are a list of lists of elements.

sg.theme('BlueMono')

# Column layout
col = [[sg.Text('col Row 1', text_color='white', background_color='blue')],
       [sg.Text('col Row 2', text_color='white', background_color='blue'), sg.Input('col input
       [sg.Text('col Row 3', text_color='white', background_color='blue'), sg.Input('col input

layout = [[sg.Listbox(values=('Listbox Item 1', 'Listbox Item 2', 'Listbox Item 3'), select_mo
          [sg.Input('Last input')],
          [sg.OK()]]

# Display the Window and get values

event, values = sg.Window('Compact 1-line Window with column', layout).Read()

sg.popup(event, values, line_width=200)
```

Recipe - Persistent Window With Text Element Updates

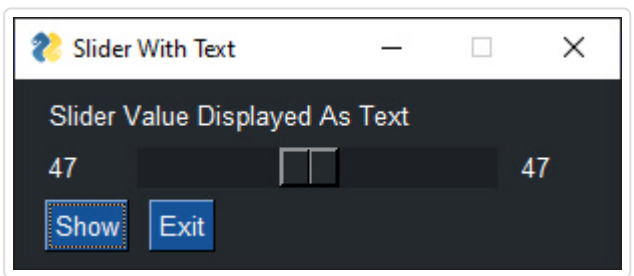
This simple program keep a window open, taking input values until the user terminates the program using the "X" button.

This Recipe has a number of concepts. * Element name aliases - `Txt` and `In` are used in the layout * Bind return key so that rather than clicking "Calculate" button, the user presses return key * No exit/close button. The window is closed using the "X" * Dark Green 7 theme - there are some nice themes, try some out for yourself * try/except for cathing errors with the floating point * These should be used sparingly * Don't place a try/except around your whole event loop to try and fix your coding errors * Displaying results using a Text element - Note: be sure and set the size to a large enough value




```
import PySimpleGUI as sg

sg.theme('Dark Green 7')

layout = [ [sg.Txt('Enter values to calculate')],
           [sg.In(size=(8,1), key='-NUMERATOR-')],
           [sg.Txt('_'  * 10)],
           [sg.In(size=(8,1), key='-DENOMINATAOR-')],
           [sg.Txt(size=(8,1), key='-OUTPUT-')  ],
           [sg.Button('Calculate', bind_return_key=True)]]

window = sg.Window('Math', layout)

while True:
    event, values = window.read()

    if event != sg.WIN_CLOSED:
        try:
            numerator = float(values['-NUMERATOR-'])
            denominator = float(values['-DENOMINATAOR-'])
            calc = numerator/denominator
        except:
            calc = 'Invalid'

        window['-OUTPUT-'].update(calc)
    else:
        break
```

Recipe - One Element Updating Another - Compound Elements



You can easily build "compound elements" in a single like of code. This recipe shows you how to add a numeric value onto a slider.

```
import PySimpleGUI as sg

layout = [[sg.Text('Slider Demonstration'), sg.Text('', key='-OUTPUT-')],
          [sg.T('0',size=(4,1), key='-LEFT-'),
           sg.Slider((0,100), key='-SLIDER-', orientation='h', enable_events=True, disable_number_
           sg.T('0', size=(4,1), key='-RIGHT-')],
          [sg.Button('Show'), sg.Button('Exit')]]

window = sg.Window('Window Title', layout)

while True:             # Event Loop
    event, values = window.read()
    print(event, values)
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    window['-LEFT-'].update(int(values['-SLIDER-']))
    window['-RIGHT-'].update(int(values['-SLIDER-']))
    if event == 'Show':
        sg.popup(f'The slider value = {values["-SLIDER-"]}')
window.close()
```

Recipe - Multiple Windows

There are ***numerous Demo Programs*** that show a multitude of techniques for running multiple windows in PySimpleGUI. Over the years these techniques have evolved. It's best to check with the Demo Propgrams as they are updated more frequently than this Cookbook.

This recipe is a design pattern for multiple windows where the first window is not active while the second window is showing. The first window is hidden to discourage continued interaction.

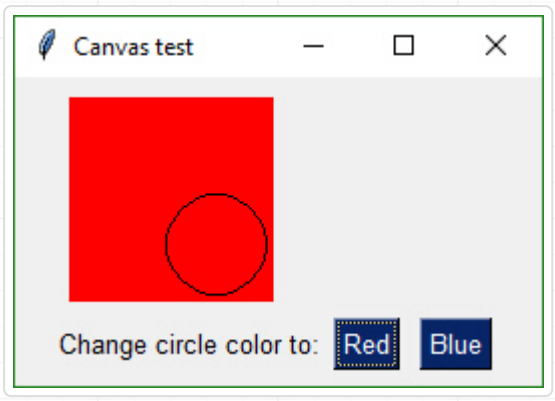
```
"""
 PySimpleGUI The Complete Course Lesson 7 - Multiple Windows"""
import PySimpleGUI as sg

# Design pattern 1 - First window does not remain active

layout = [[ sg.Text('Window 1'),],
          [sg.Input()],
          [sg.Text('', key='_OUTPUT_')],
          [sg.Button('Launch 2')]]

win1 = sg.Window('Window 1', layout)
win2_active=False
while True:
    ev1, vals1 = win1.Read(timeout=100)
    if ev1 == sg.WIN_CLOSED:
        break
    win1['_OUTPUT_'].update(vals1[0])

    if ev1 == 'Launch 2'  and not win2_active:
        win2_active = True
        win1.Hide()
        layout2 = [[sg.Text('Window 2')],       # note must create a layout from scratch every tim
            [sg.Button('Exit')]]

        win2 = sg.Window('Window 2', layout2)
        while True:
            ev2, vals2 = win2.Read()
            if ev2 == sg.WIN_CLOSED or ev2 == 'Exit':
                win2.Close()
                win2_active = False
                win1.UnHide()
                break
```

tkinter Canvas Widget

The Canvas Element is one of the few tkinter objects that are directly accessible. The tkinter Canvas widget itself can be retrieved from a Canvas Element like this:

```
    can = sg.Canvas(size=(100,100))
    tkcanvas = can.TKCanvas
    tkcanvas.create_oval(50, 50, 100, 100)
```

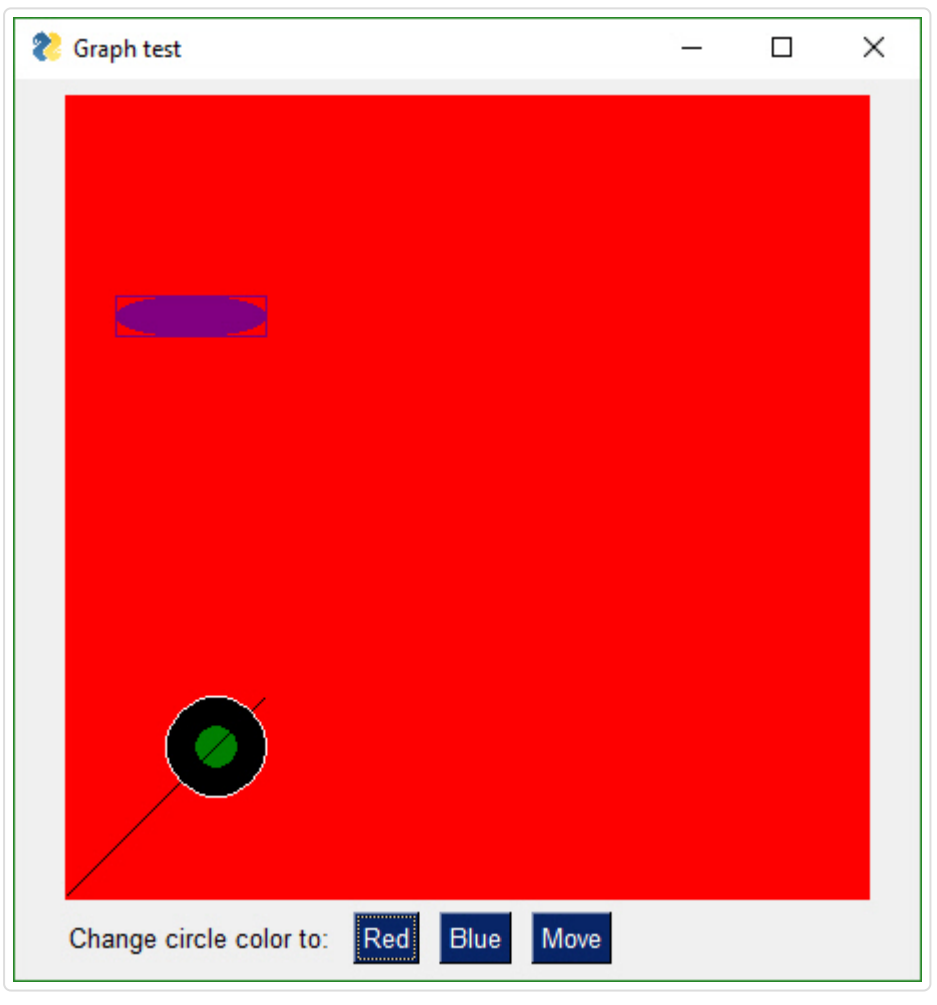
While it's fun to scribble on a Canvas Widget, try Graph Element makes it a downright pleasant experience. You do not have to worry about the tkinter coordinate system and can instead work in your own coordinate system.




```
import PySimpleGUI as sg

layout = [
    [sg.Canvas(size=(100, 100), background_color='red', key= 'canvas')],
    [sg.T('Change circle color to:'), sg.Button('Red'), sg.Button('Blue')]
    ]

window = sg.Window('Canvas test', layout, finalize=True)

canvas = window['canvas']
cir = canvas.TKCanvas.create_oval(50, 50, 100, 100)

while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED:
        break
    if event == 'Blue':
        canvas.TKCanvas.itemconfig(cir, fill="Blue")
    elif event == 'Red':
        canvas.TKCanvas.itemconfig(cir, fill="Red")
```

Graph Element - drawing circle, rectangle, etc, objects

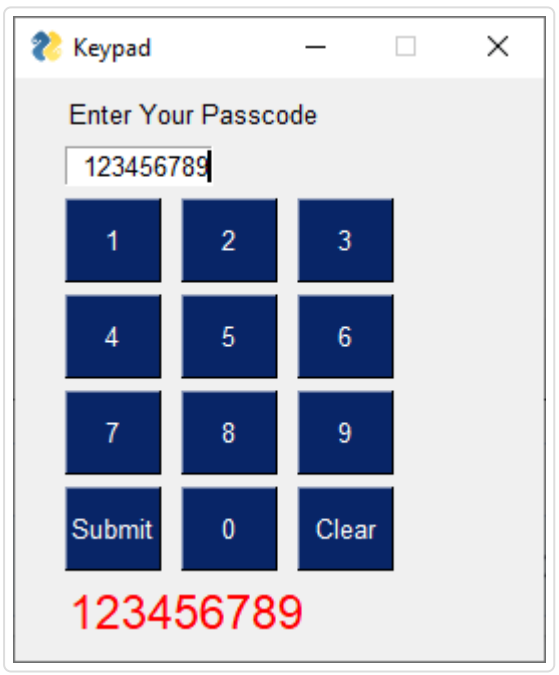
Just like you can draw on a tkinter widget, you can also draw on a Graph Element. Graph Elements are easier on the programmer as you get to work in your own coordinate system.



```
import PySimpleGUI as sg

layout = [
          [sg.Graph(canvas_size=(400, 400), graph_bottom_left=(0,0), graph_top_right=(400, 40
          [sg.T('Change circle color to:'), sg.Button('Red'), sg.Button('Blue'), sg.Button('M
          ]

window = sg.Window('Graph test', layout, finalize=True)

graph = window['graph')
circle = graph.DrawCircle((75,75), 25, fill_color='black',line_color='white')
point = graph.DrawPoint((75,75), 10, color='green')
oval = graph.DrawOval((25,300), (100,280), fill_color='purple', line_color='purple'  )
rectangle = graph.DrawRectangle((25,300), (100,280), line_color='purple'  )
line = graph.DrawLine((0,0), (100,100))

while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED:
        break
    if event is 'Blue':
        graph.TKCanvas.itemconfig(circle, fill = "Blue")
    elif event is 'Red':
        graph.TKCanvas.itemconfig(circle, fill = "Red")
    elif event is 'Move':
        graph.MoveFigure(point, 10,10)
        graph.MoveFigure(circle, 10,10)
        graph.MoveFigure(oval, 10,10)
        graph.MoveFigure(rectangle, 10,10)
```

Keypad Touchscreen Entry - Input Element Update

This Recipe implements a Raspberry Pi touchscreen based keypad entry. As the digits are entered using the buttons, the Input Element above it is updated with the input digits.
There are a number of features used in this Recipe including:
Default Element Size
auto_size_buttons
Button
Dictionary Return values
* Update of Elements in window (Input, Text)



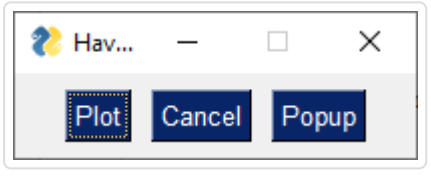
```
import PySimpleGUI as sg

layout = [[sg.Text('Enter Your Passcode')],
          [sg.Input(size=(10, 1), justification='right', key='input')],
          [sg.Button('1'), sg.Button('2'), sg.Button('3')],
          [sg.Button('4'), sg.Button('5'), sg.Button('6')],
          [sg.Button('7'), sg.Button('8'), sg.Button('9')],
          [sg.Button('Submit'), sg.Button('0'), sg.Button('Clear')],
          [sg.Text(size=(15, 1), font=('Helvetica', 18), text_color='red', key='out')]]

window = sg.Window('Keypad', layout, default_button_element_size=(5,2), auto_size_buttons=False)

# Loop forever reading the window's values, updating the Input field
keys_entered = ''
while True:
    event, values = window.read()  # read the window
    if event == sg.WIN_CLOSED:  # if the X button clicked, just exit
        break
    if event == 'Clear':  # clear keys if clear button
        keys_entered = ''
    elif event in '1234567890':
        keys_entered = values['input']  # get what's been entered so far
        keys_entered += event  # add the new digit
    elif event == 'Submit':
        keys_entered = values['input']
        window['out'].update(keys_entered)  # output the final string

    window['input'].update(keys_entered)  # change the window to reflect current key string
```

Matplotlib Window With GUI Window

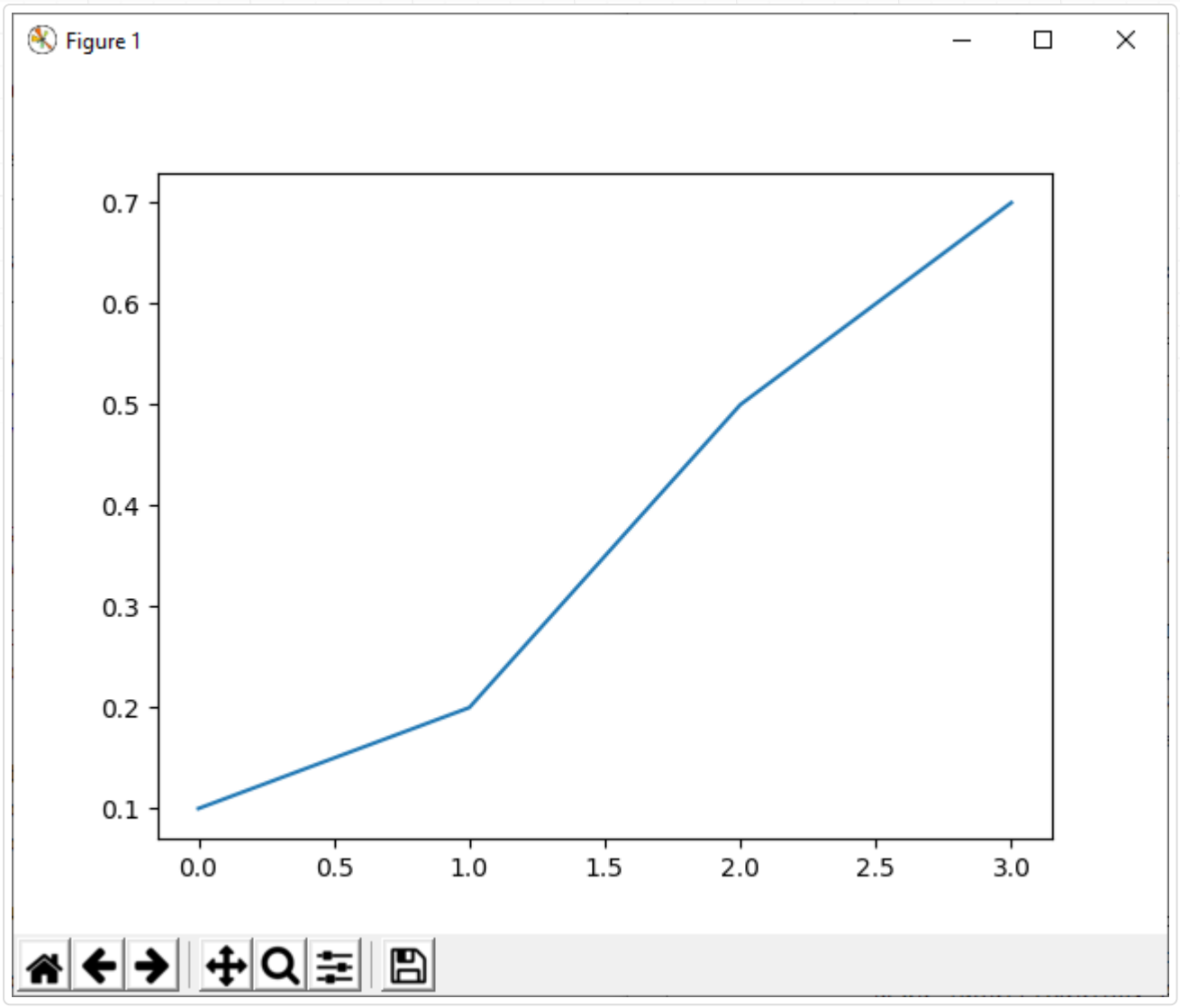
There are 2 ways to use PySimpleGUI with Matplotlib. Both use the standard tkinter based Matplotlib.

The simplest is when both the interactive Matplotlib window and a PySimpleGUI window are running at the same time.

First the PySimpleGUI window appears giving you 3 options.



Clicking "Plot" will create the Matplotlib window

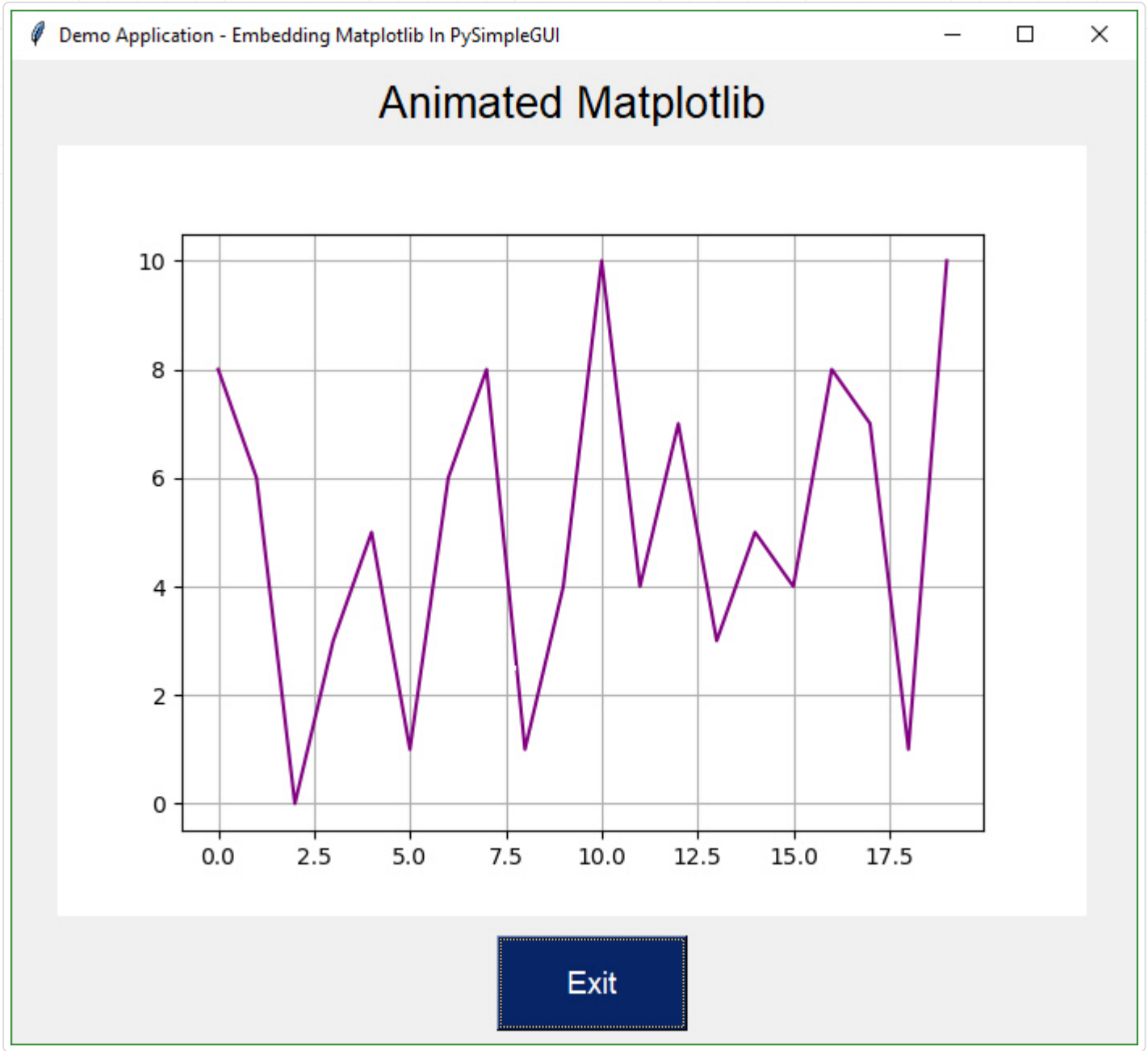


You can click the "Popup" button in the PySimpleGUI window and you'll see a popup window, proving the your GUI is still alive and operational.

```
import PySimpleGUI as sg
import matplotlib.pyplot as plt

"""
    Simultaneous PySimpleGUI Window AND a Matplotlib Interactive Window
    A number of people have requested the ability to run a normal PySimpleGUI window that
    launches a MatplotLib window that is interactive with the usual Matplotlib controls.
    It turns out to be a rather simple thing to do.  The secret is to add parameter block=False to
"""

def draw_plot():
    plt.plot([0.1, 0.2, 0.5, 0.7])
    plt.show(block=False)

layout = [[sg.Button('Plot'), sg.Cancel(), sg.Button('Popup')]]

window = sg.Window('Have some Matplotlib....', layout)

while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, 'Cancel'):
        break
    elif event == 'Plot':
        draw_plot()
    elif event == 'Popup':
        sg.popup('Yes, your application is still running')
window.close()
```

Animated Matplotlib Graph

Use the Canvas Element to create an animated graph. The code is a bit tricky to follow, but if you know Matplotlib then this recipe shouldn't be too difficult to copy and modify.


Demo Application - Embedding Matplotlib In PySimpleGUI
Animated Matplotlib
10
8
6
4
2
0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Exit

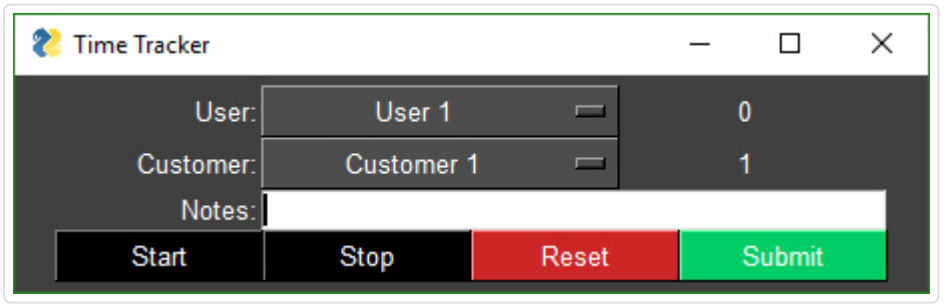
```
from tkinter import *
from random import randint
import PySimpleGUI as sg
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, FigureCanvasAgg
from matplotlib.figure import Figure
import matplotlib.backends.tkagg as tkagg
import tkinter as Tk

fig = Figure()

ax = fig.add_subplot(111)
ax.set_xlabel("X axis")
ax.set_ylabel("Y axis")
ax.grid()

layout = [[sg.Text('Animated Matplotlib', size=(40, 1), justification='center', font='Helvetica 20
          [sg.Canvas(size=(640, 480), key='canvas')],
          [sg.Button('Exit', size=(10, 2), pad=((280, 0), 3), font='Helvetica 14')]]

# create the window and show it without the plot


window = sg.Window('Demo Application - Embedding Matplotlib In PySimpleGUI', layout, finalize=True
# needed to access the canvas element prior to reading the window

canvas_elem = window['canvas']

graph = FigureCanvasTkAgg(fig, master=canvas_elem.TKCanvas)
canvas = canvas_elem.TKCanvas

dpts = [randint(0, 10) for x in range(10000)]
# Our event loop
for i in range(len(dpts)):
    event, values = window.read(timeout=20)
    if event == 'Exit' or event == sg.WIN_CLOSED:
        exit(69)

    ax.cla()
    ax.grid()

    ax.plot(range(20), dpts[i:i + 20], color='purple')
    graph.draw()
    figure_x, figure_y, figure_w, figure_h = fig.bbox.bounds
    figure_w, figure_h = int(figure_w), int(figure_h)
    photo = Tk.PhotoImage(master=canvas, width=figure_w, height=figure_h)

    canvas.create_image(640 / 2, 480 / 2, image=photo)

    figure_canvas_agg = FigureCanvasAgg(fig)
    figure_canvas_agg.draw()

    tkagg.blit(photo, figure_canvas_agg.get_renderer()._renderer, colormode=2)
```

Tight Layout with Button States

Saw this example layout written in tkinter and liked it so much I duplicated the interface. It's "tight", clean, and has a nice dark look and feel.

This Recipe also contains code that implements the button interactions so that you'll have a template to build from.

In other GUI frameworks this program would be most likely "event driven" with callback functions being used to communicate button events. The "event loop" would be handled by the GUI engine. If code already existed that used a call-back mechanism, the loop in the example code below could simply call these callback functions directly based on the button text it receives in the window.read call.

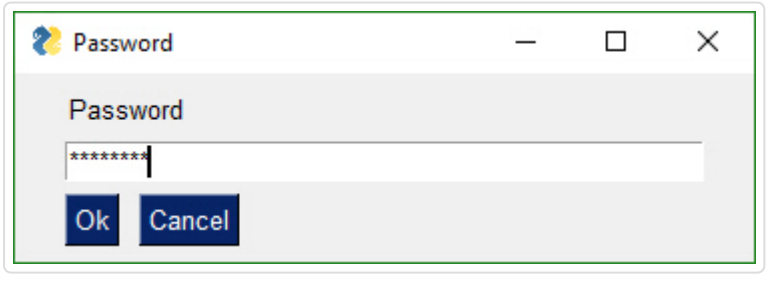


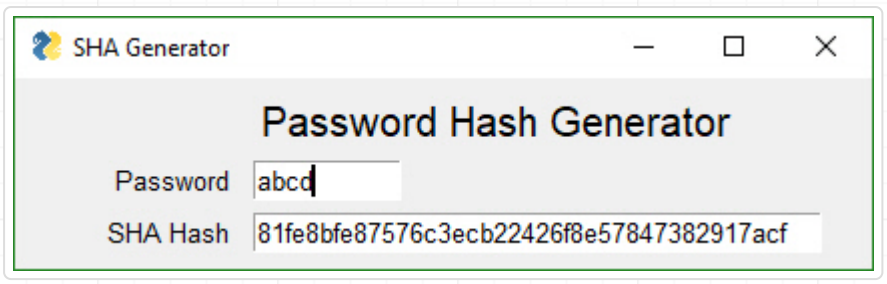
```
import PySimpleGUI as sg
"""
Demonstrates using a "tight" layout with a Dark theme.
Shows how button states can be controlled by a user application.  The program manages the disa
states for buttons and changes the text color to show greyed-out (disabled) buttons
"""

sg.theme('Dark')
sg.set_options(element_padding=(0,0))

layout = [[sg.T('User:', pad=((3,0),0)), sg.OptionMenu(values = ('User 1', 'User 2'), size=(20
          [sg.T('Customer:', pad=((3,0),0)), sg.OptionMenu(values=('Customer 1', 'Customer 2')
          [sg.T('Notes:', pad=((3,0),0)), sg.In(size=(44,1), background_color='white', text_co
          [sg.Button('Start', button_color=('white', 'black'), key='Start'),
           sg.Button('Stop', button_color=('white', 'black'), key='Stop'),
           sg.Button('Reset', button_color=('white', 'firebrick3'), key='Reset'),
           sg.Button('Submit', button_color=('white', 'springgreen4'), key='Submit')]
          ]

window = sg.Window("Time Tracker", layout, default_element_size=(12,1), text_justification='r'

window['Stop'].update(disabled=True)
window['Reset'].update(disabled=True)
window['Submit'].update(disabled=True)
recording = have_data = False
while True:
    event, values = window.read()
    print(event)
    if event == sg.WIN_CLOSED:
        exit(69)
    if event is 'Start':
        window['Start'].update(disabled=True)
        window['Stop'].update(disabled=False)
        window['Reset'].update(disabled=False)
        window['Submit'].update(disabled=True)
        recording = True
    elif event is 'Stop'  and recording:
        window['Stop'].update(disabled=True)
        window['Start'].update(disabled=False)
        window['Submit'].update(disabled=False)
        recording = False
        have_data = True
    elif event is 'Reset':
        window['Stop'].update(disabled=True)
        window['Start'].update(disabled=False)
        window['Submit'].update(disabled=True)
        window['Reset'].update(disabled=False)
        recording = False
        have_data = False
    elif event is 'Submit'  and have_data:
        window['Stop'].update(disabled=True)
        window['Start'].update(disabled=False)
        window['Submit'].update(disabled=True)
        window['Reset'].update(disabled=False)
        recording = False
```

Password Protection For Scripts

You get 2 scripts in one.

Use the upper half to generate your hash code. Then paste it into the code in the lower half. Copy and paste lower 1/2 into your code to get password protection for your script without putting the password into your source code.






```
import PySimpleGUI as sg
import hashlib

'''
    Create a secure login for your scripts without having to include your password in the program
    1. Choose a password
    2. Generate a hash code for your chosen password by running program and entering 'gui' as the
    3. Type password into the GUI
    4. Copy and paste hash code Window GUI into variable named login_password_hash
    5. Run program again and test your login!
'''

# Use this GUI to get your password's hash code
def HashGeneratorGUI():
    layout = [[sg.T('Password Hash Generator', size=(30,1), font='Any 15')],
              [sg.T('Password'), sg.In(key='password')],
              [sg.T('SHA Hash'), sg.In('', size=(40,1), key='hash')],
              ]

    window = sg.Window('SHA Generator', layout, auto_size_text=False, default_element_size=(10,1),
                       text_justification='r', return_keyboard_events=True, grab_anywhere=False)


    while True:
        event, values = window.read()
        if event ==  sg.WIN_CLOSED:
              exit(69)

        password = values['password']
        try:
            password_utf = password.encode('utf-8')
            sha1hash = hashlib.sha1()
            sha1hash.update(password_utf)
            password_hash = sha1hash.hexdigest()
            window['hash'].update(password_hash)
        except:
            pass

# ------------------------------ Paste this code into your program / script ----------------------
# determine if a password matches the secret password by comparing SHA1 hash codes
def PasswordMatches(password, hash):
    password_utf = password.encode('utf-8')
    sha1hash = hashlib.sha1()
    sha1hash.update(password_utf)
    password_hash = sha1hash.hexdigest()
    if password_hash == hash:
        return True
    else:
        return False

login_password_hash = '5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8'
password = sg.popup_get_text('Password', password_char='*')
if password == 'gui':                # Remove when pasting into your program
    HashGeneratorGUI()               # Remove when pasting into your program
    exit(69)                         # Remove when pasting into your program
if PasswordMatches(password, login_password_hash):
    print('Login SUCCESSFUL')
else:
    print('Login FAILED!!')
```

Desktop Floating Toolbar

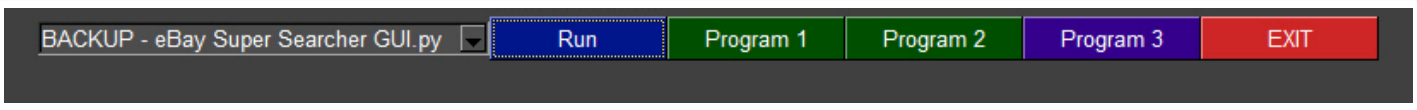
Hiding your windows commmand window

If you would like your python program to run without showing a console window, then you can name your file with a `.pyw` extension and open the file using `pythonw` instead of `python`. This is the preferred way to launch a final version of a PySimpleGUI program as there is no need for a Console window and it will be a much more "Windows-like" experience.

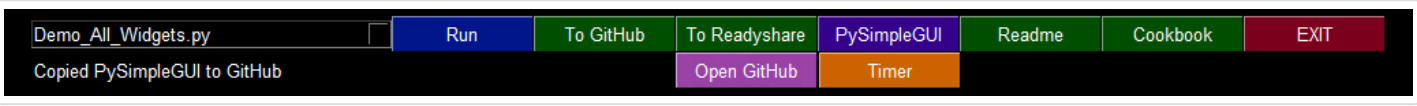
Floating toolbar

NOTE - Please look in the Demo Programs that use the newer Exec APIs. This Recipe uses an older technique to launch subprocesses. You'll find the Exec APIs documented in the main documentation and the Call Reference.

"Launchers" have come a long long ways since the early days of PySimpleGUI. This recipe directly calls `subprocess.Popen`. In 2020 a new set of APIs, the Exec APIs, were added to PySimpleGUI. These calls simplify the subprocess calls thus making them more approachable for newcomers to Python.



You can easily change colors to match your background by changing a couple of parameters in the code.

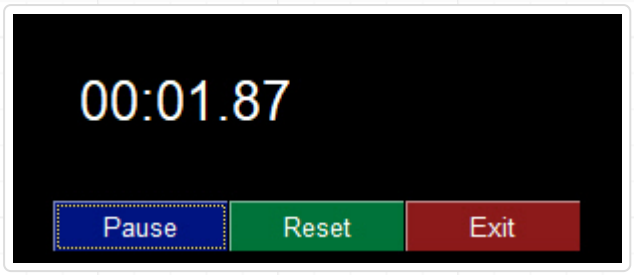


```
import PySimpleGUI as sg
import subprocess
import os
import sys

"""
 Demo_Toolbar - A floating toolbar with quick launcher     One cool PySimpleGUI demo. Shows bo
 You can setup a specific program to launch when a button is clicked, or use the Combobox to s

ROOT_PATH = './'

def Launcher():

    def print(line):
        window['output'].update(line)

    sg.theme('Dark')

    namesonly = [f for f in os.listdir(ROOT_PATH) if f.endswith('.py') ]

    sg.set_options(element_padding=(0,0), button_element_size=(12,1), auto_size_buttons=False)
    layout =  [[sg.Combo(values=namesonly, size=(35,30), key='demofile'),
                sg.Button('Run', button_color=('white', '#00168B')),
                sg.Button('Program 1'),
                sg.Button('Program 2'),
                sg.Button('Program 3', button_color=('white', '#35008B')),
                sg.Button('EXIT', button_color=('white','firebrick3'))],
               [sg.T('', text_color='white', size=(50,1), key='output')]]

    window = sg.Window('Floating Toolbar', layout, no_titlebar=True, keep_on_top=True)

    # ---======-- Loop taking in user input (events) --- #
    while True:
        (event, value) = window.read()
        if event ==  'EXIT'  or event == sg.WIN_CLOSED:
            break # exit button clicked
        if event ==  'Program 1':
            print('Run your program 1 here!')
        elif event ==  'Program 2':
            print('Run your program 2 here!')
        elif event ==  'Run':
            file = value['demofile']
            print('Launching %s'%file)
            ExecuteCommandSubprocess('python', os.path.join(ROOT_PATH, file))
        else:
            print(event)

def ExecuteCommandSubprocess(command, *args, wait=False):
    try:
        if sys.platwindow == 'linux':
            arg_string = ''
            for arg in args:
                 arg_string += ' '  + str(arg)
            sp = subprocess.Popen(['python3'  + arg_string, ], shell=True, stdout=subprocess.P
        else:
            sp = subprocess.Popen([command, list(args)], shell=True, stdout=subprocess.PIPE, s
        if wait:
            out, err = sp.communicate()
            if out:
                print(out.decode("utf-8"))
            if err:
                print(err.decode("utf-8"))
    except: pass



if __name__ == '__main__':
    Launcher()
```

Desktop Floating Widget - Timer

This is a little widget you can leave running on your desktop. Will hopefully see more of these for things like checking email, checking server pings, displaying system information, dashboards, etc .

Much of the code is handling the button states in a fancy way. It could be much simpler if you don't change the button text based on state.

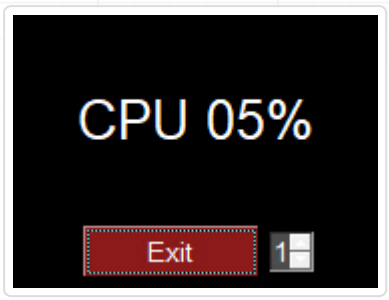



```
import sys
if sys.version_info[0] >= 3:
    import PySimpleGUI as sg
else:
    import PySimpleGUI27 as sg
import time

"""
 Timer Desktop Widget Creates a floating timer that is always on top of other windows You move it

 While the timer ticks are being generated by PySimpleGUI's "timeout" mechanism, the actual value
  of the timer that is displayed comes from the system timer, time.time().  This guarantees an
  accurate time value is displayed regardless of the accuracy of the PySimpleGUI timer tick. If
  this design were not used, then the time value displayed would slowly drift by the amount of tim
  it takes to execute the PySimpleGUI read and update calls (not good!)

 NOTE - you will get a warning message printed when you exit using exit button.
 It will look something like: invalid command name \"1616802625480StopMove\"
"""


# ----------------  Create Form  ----------------
sg.theme('Black')
sg.set_options(element_padding=(0, 0))

layout = [[sg.Text('')],
          [sg.Text('', size=(8, 2), font=('Helvetica', 20), justification='center', key='text')],
          [sg.Button('Pause', key='button', button_color=('white', '#001480')),
           sg.Button('Reset', button_color=('white', '#007339'), key='Reset'),
           sg.Exit(button_color=('white', 'firebrick4'), key='Exit')]]

window = sg.Window('Running Timer', layout, no_titlebar=True, auto_size_buttons=False, keep_on_top

# ----------------  main loop  ----------------
current_time = 0
paused = False
start_time = int(round(time.time() * 100))
while (True):
    # --------- Read and update window --------
    if not paused:
        event, values = window.read(timeout=10)
        current_time = int(round(time.time() * 100)) - start_time
    else:
        event, values = window.read()
    if event == 'button':
        event = window[event).GetText()
    # --------- Do Button Operations --------
    if event == sg.WIN_CLOSED or event == 'Exit':        # ALWAYS give a way out of program
        break
    if event is 'Reset':
        start_time = int(round(time.time() * 100))
        current_time = 0
        paused_time = start_time
    elif event == 'Pause':
        paused = True
        paused_time = int(round(time.time() * 100))
        element = window['button')
        element.update(text='Run')
    elif event == 'Run':
        paused = False
        start_time = start_time + int(round(time.time() * 100)) - paused_time
        element = window['button')
        element.update(text='Pause')

    # --------- Display timer in window --------
    window['text'].update('{:02d}:{:02d}.{:02d}'.format((current_time // 100) // 60,
                                                        (current_time // 100) % 60,
                                                        current_time % 100))
```

Desktop Floating Widget - CPU Utilization

Like the Timer widget above, this script can be kept running. You will need the package psutil installed in order to run this Recipe.

The spinner changes the number of seconds between reads. Note that you will get an error message printed when exiting because the window does not have have a titlebar. It's a known problem.



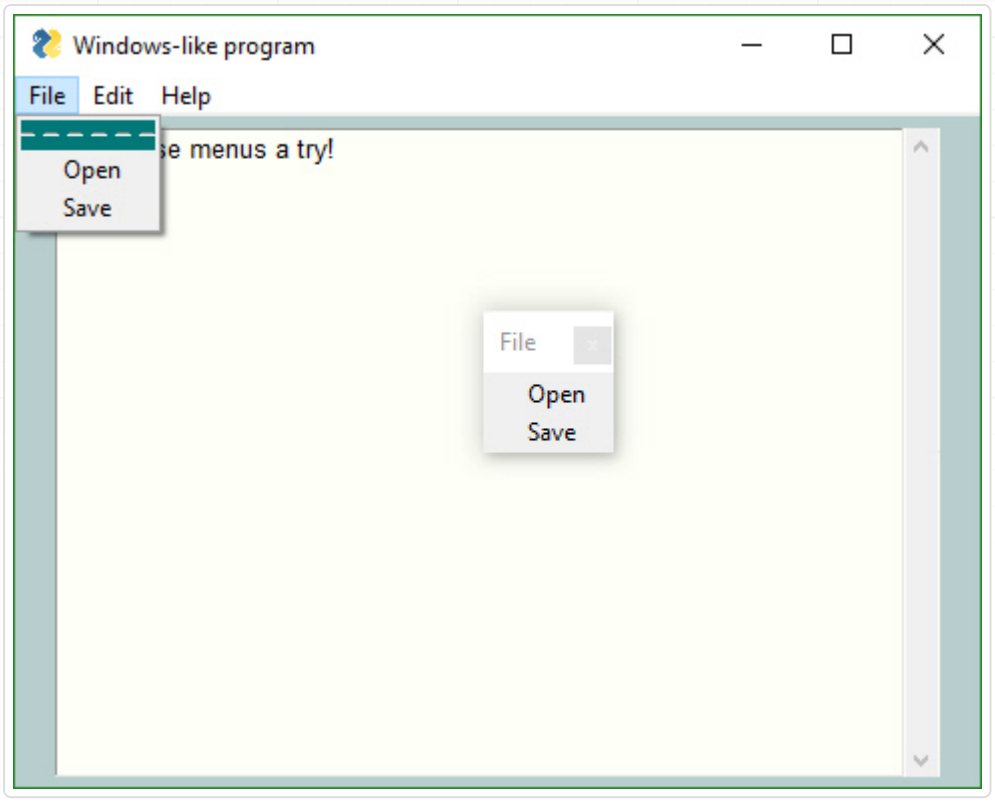

```
import PySimpleGUI as sg
import psutil

# ----------------  Create Window  ----------------
sg.theme('Black')
layout = [[sg.Text('')],
          [sg.Text('', size=(8, 2), font=('Helvetica', 20), justification='center', key='text')],
          [sg.Exit(button_color=('white', 'firebrick4'), pad=((15, 0), 0)),
           sg.Spin([x + 1 for x in range(10)], 1, key='spin')]]

window = sg.Window('Running Timer', layout, no_titlebar=True, auto_size_buttons=False, keep_on_top
                   grab_anywhere=True)

# ----------------  main loop  ----------------
while (True):
    # --------- Read and update window --------
    event, values = window.read(timeout=0)

    # --------- Do Button Operations --------
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    try:
        interval = int(values['spin'])
    except:
        interval = 1

    cpu_percent = psutil.cpu_percent(interval=interval)

    # --------- Display timer in window --------

    window['text'].update(f'CPU {cpu_percent:02.0f}%')

# Broke out of main loop. Close the window.
window.close()
```

Menus

Menus are nothing more than buttons that live in a menu-bar. When you click on a menu item, you get back a "button" with that menu item's text, just as you would had that text been on a button.

Menu's are defined separately from the GUI window. To add one to your window, simply insert sg.Menu(menu_layout). The menu definition is a list of menu choices and submenus. They are a list of lists. Copy the Recipe and play with it. You'll eventually get when you're looking for.

If you double click the dashed line at the top of the list of choices, that menu will tear off and become a floating toolbar. How cool! To enable this feature, set the parameter `tearoff=True` in your call to `sg.Menu()`

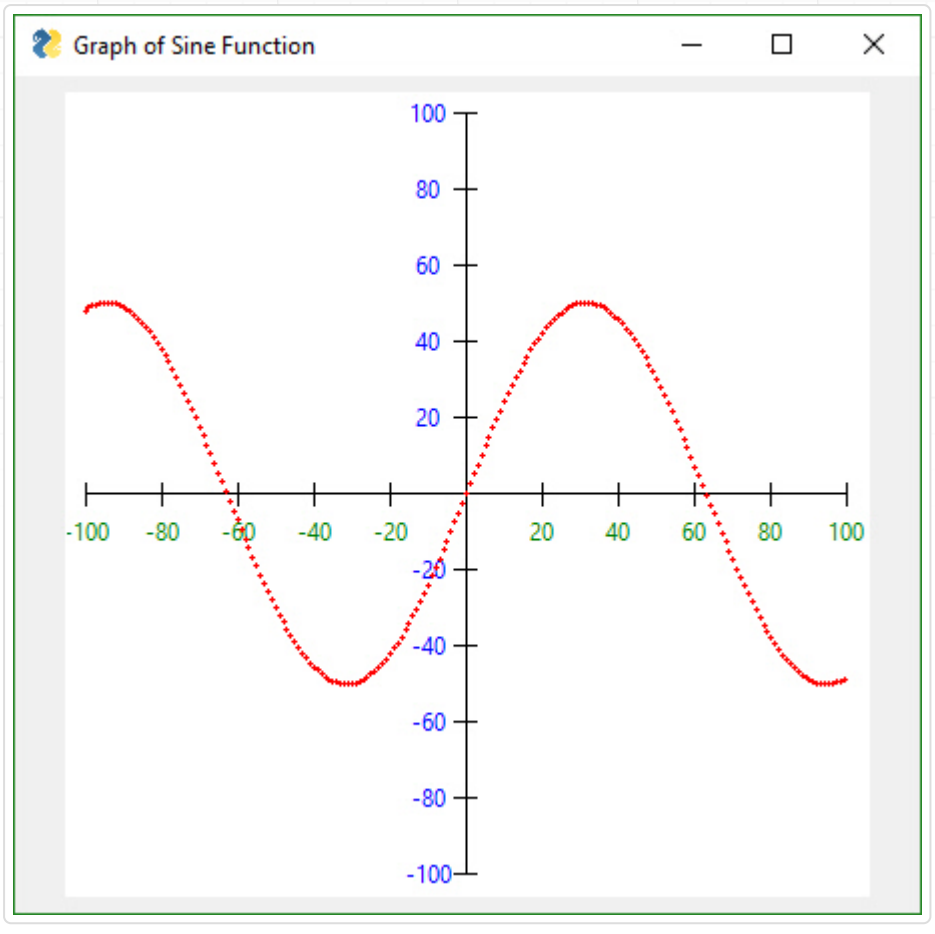



```
import PySimpleGUI as sg

sg.theme('LightGreen')
sg.set_options(element_padding=(0, 0))

# ------ Menu Definition ------ #
menu_def = [['File', ['Open', 'Save', 'Exit'  ]],
            ['Edit', ['Paste', ['Special', 'Normal', ], 'Undo'], ],
            ['Help', 'About...'], ]

# ------ GUI Defintion ------ #
layout = [
    [sg.Menu(menu_def, )],
    [sg.Output(size=(60, 20))]
         ]

window = sg.Window("Windows-like program", layout, default_element_size=(12, 1), auto_size_tex
                   default_button_element_size=(12, 1))

# ------ Loop & Process button menu choices ------ #
while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Exit':
        break
    print('Button = ', event)
    # ------ Process menu choices ------ #
    if event == 'About...':
        sg.popup('About this program', 'Version 1.0', 'PySimpleGUI rocks...')
    elif event == 'Open':
        filename = sg.popup_get_file('file to open', no_window=True)
        print(filename)
```

Graphing with Graph Element

Use the Graph Element to draw points, lines, circles, rectangles using ***your*** coordinate systems rather than the underlying graphics coordinates.

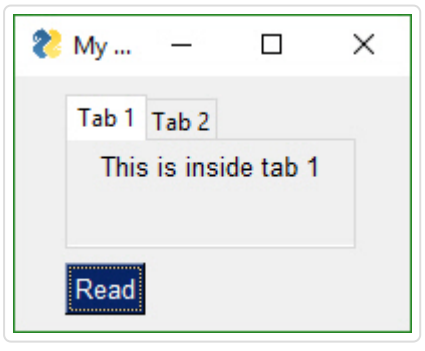
In this example we're defining our graph to be from -100, -100 to +100,+100. That means that zero is in the middle of the drawing. You define this graph description in your call to Graph.




```
import math
import PySimpleGUI as sg

layout = [[sg.Graph(canvas_size=(400, 400), graph_bottom_left=(-105,-105), graph_top_right=(105,10

window = sg.Window('Graph of Sine Function', layout, grab_anywhere=True, finalize=True)
graph = window['graph']

# Draw axis
graph.DrawLine((-100,0), (100,0))
graph.DrawLine((0,-100), (0,100))

for x in range(-100, 101, 20):
    graph.DrawLine((x,-3), (x,3))
    if x != 0:
        graph.DrawText( x, (x,-10), color='green')

for y in range(-100, 101, 20):
    graph.DrawLine((-3,y), (3,y))
    if y != 0:
        graph.DrawText( y, (-10,y), color='blue')

# Draw Graph
for x in range(-100,100):
    y = math.sin(x/20)*50
    graph.DrawCircle((x,y), 1, line_color='red', fill_color='red')

event, values = window.read()
```

Tabs

Tabs bring not only an extra level of sophistication to your window layout, they give you extra room to add more elements. Tabs are one of the 3 container Elements, Elements that hold or contain other Elements. The other two are the Column and Frame Elements.



```
import PySimpleGUI as sg

tab1_layout =  [[sg.T('This is inside tab 1')]]

tab2_layout = [[sg.T('This is inside tab 2')],
               [sg.In(key='in')]]

layout = [[sg.TabGroup([[sg.Tab('Tab 1', tab1_layout, tooltip='tip'), sg.Tab('Tab 2', tab2_layout)
          [sg.Button('Read')]]

window = sg.Window('My window with tabs', layout, default_element_size=(12,1))

while True:
    event, values = window.read()
    print(event,values)
    if event == sg.WIN_CLOSED:           # always,  always give a way out!
        break
```

Creating a Windows .EXE File

NEW in 2021 was the release of the `psgcompiler` project. This is a front-end to the popular PyInstaller package. `psgcompiler` adds a PySimpleGUI GUI front-end to PyInstaller, making it easier for you to create binary versions of your code for distributing to people that do not have Python stalled on their system.

The `psgcompiler` will eventually support multiple back-ends. At the moment, only PyInstaller is supported. A HUGE "THANK YOU" to the PyInstaller project for making a brilliant program that enables Python programmers to share their work with non-Python users.

`psgcompiler` makes it possible to create a single .EXE file that can be distributed to Windows users. There is no requirement to install the Python interpreter on the PC you wish to run it on. Everything it needs is in the one EXE file, assuming you're running a somewhat up to date version of Windows.

You can also make APP files for the Mac and binary distributable for Linux as well. However, in order to do so, you must run the program on the OS you are targeting. In other words it is not a cross-compiler. You cannot run `psgcompiler` / `PyInstaller` on Windows and produce a Mac executable.

If you want to directly user PyInstaller instead of psgcompiler, then you can install the packages separately and use PyInstaller directly.

```
pip install PySimpleGUI
pip install PyInstaller
```

To create your EXE file from your program that uses PySimpleGUI, `my_program.py`, enter this command in your Windows command prompt:

pyinstaller -wF my_program.py

You will be left with a single file, `my_program.exe`, located in a folder named `dist` under the folder where you executed the `pyinstaller` command.

That's all... Run your `my_program.exe` file on the Windows machine of your choosing.

> "It's just that easy."
>
> (famous last words that screw up just about anything being referenced)

Your EXE file should run without creating a "shell window". Only the GUI window should show up on your taskbar.

Author & Owner

The PySimpleGUI Organization

Send correspondence to PySimpleGUI@PySimpleGUI.com prior to use of documentation

Documentation built with MkDocs.