PySimpleGUI

Now supports both Python 2.7 & 3

Announcements of Latest Developments

ReadTheDocs

COOKBOOK!

Brief Tutorial

Latest Demos and Master Branch on GitHub

Super-simple GUI to use... Powerfully customizable.

Home of the 1-line custom GUI and 1-line progress meter

Note regarding Python versions

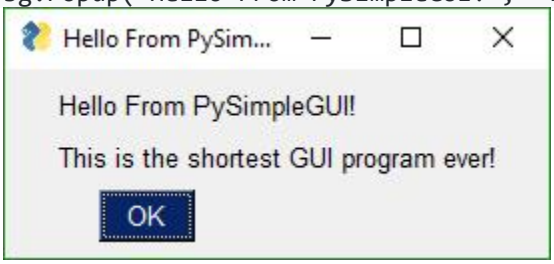
As of 9/25/2018 **both Python 3 and Python 2.7 are supported**! The Python 3 version is named `PySimpleGUI`. The Python 2.7 version is `PySimpleGUI27`. They are installed separately and the imports are different. See instructions in Installation section for more info.

Looking for a GUI package?

- Taking your Python code from the world of command lines and into the convenience of a GUI? *
- Have a Raspberry **Pi** with a touchscreen that's going to waste because you don't have the time to learn a GUI SDK?
- Into Machine Learning and are sick of the command line?
- Would like to distribute your Python code to Windows users as a single .EXE file that launches straight into a GUI, much like a WinForms app?

Look no further, **you've found your GUI package**.

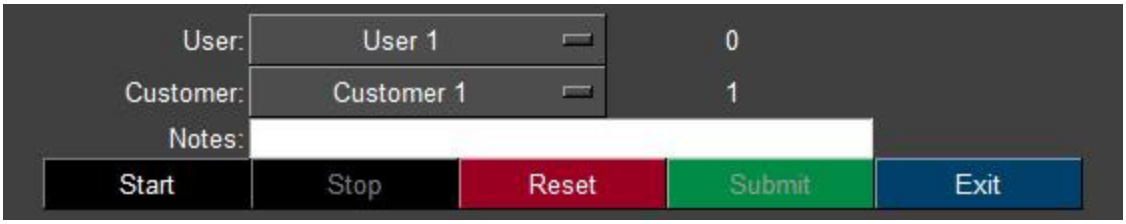
```
import PySimpleGUI as sg

sg.Popup('Hello From PySimpleGUI!', 'This is the shortest GUI program ever!')
```



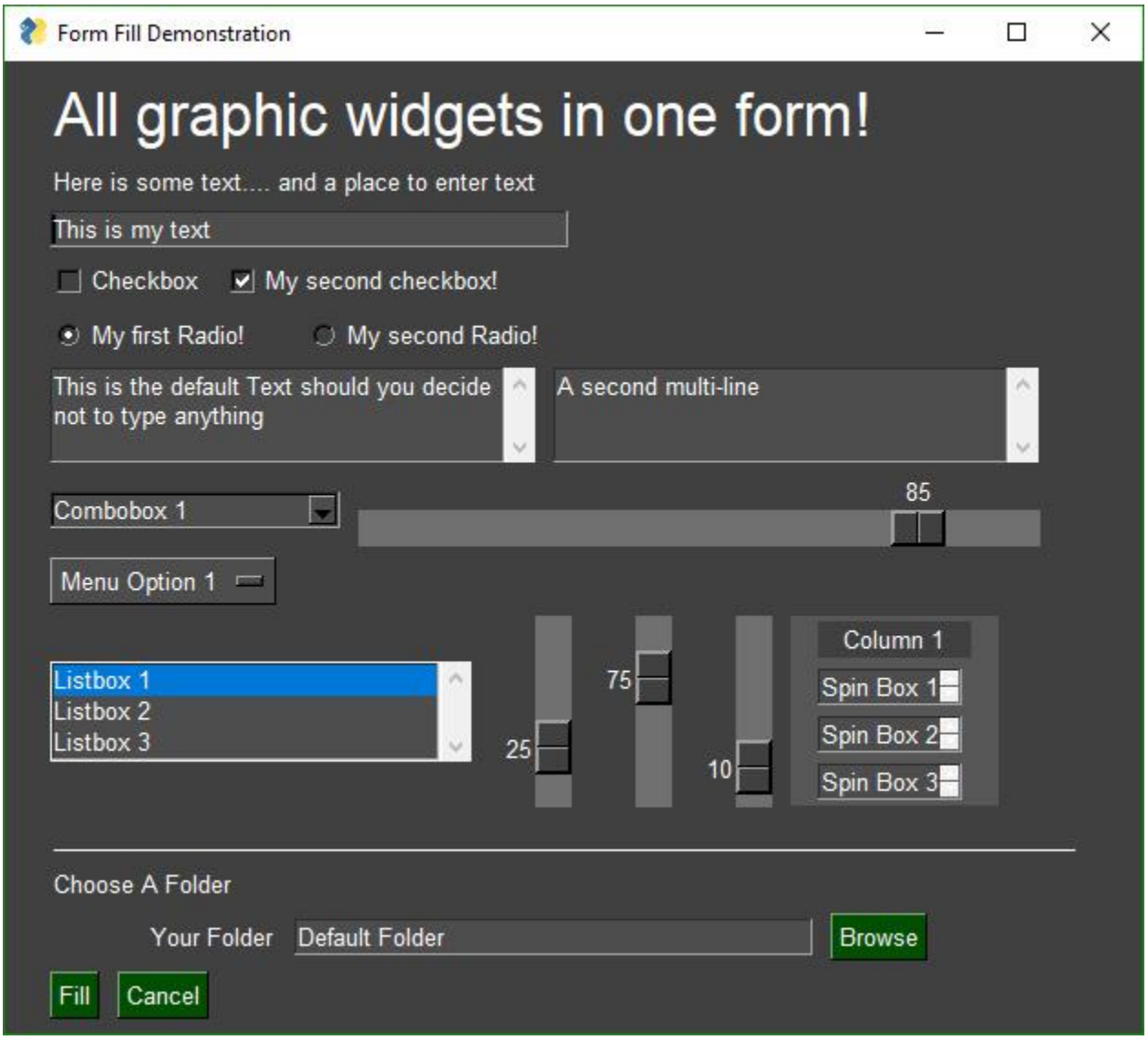
Or how about a ***custom GUI*** in 1 line of code?

```
import PySimpleGUI as sg

button, (filename,) = sg.Window('Get filename example').
LayoutAndRead([[sg.Text('Filename')], [sg.Input(), sg.FileBrowse()], [sg.OK(),
sg.Cancel()] ])
```

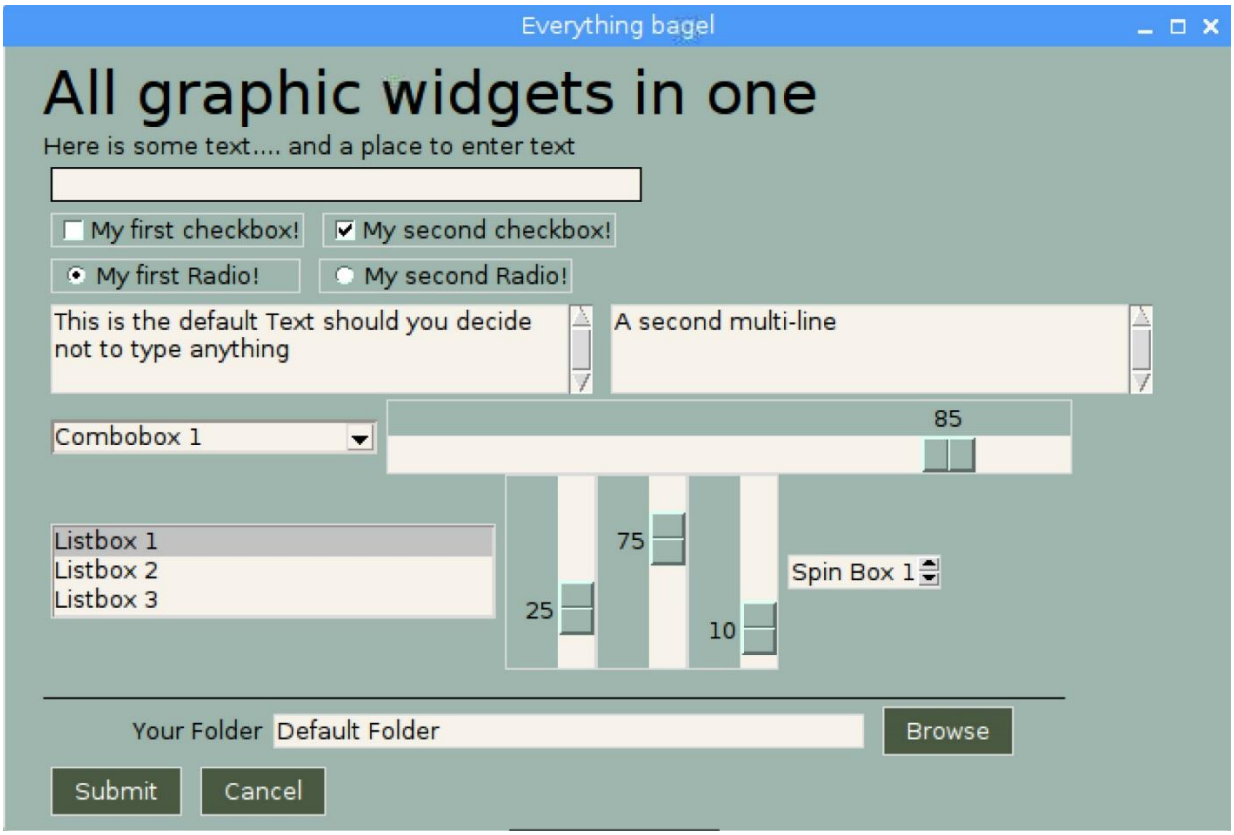
Build beautiful customized windows that fit your specific problem. Let PySimpleGUI solve your GUI problem while you solve your real problems. Look through the Cookbook, find a matching recipe, copy, paste, run within minutes. This is the process PySimpleGUI was designed to facilitate.



PySimpleGUI wraps tkinter so that you get all the same widgets as you would tkinter, but you interact with them in a more friendly way. It does the layout and boilerplate code for you and presents you with a simple, efficient interface.

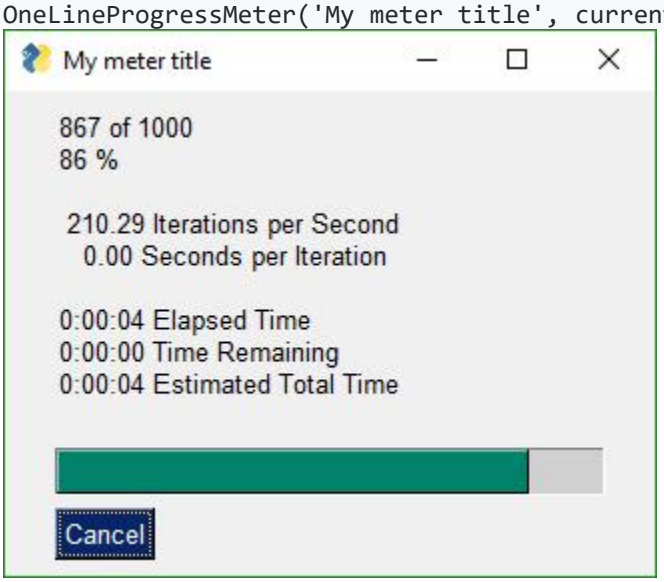


Perhaps you're looking for a way to interact with your **Raspberry Pi** in a more friendly way. The same for shown as on Pi (roughly the same)

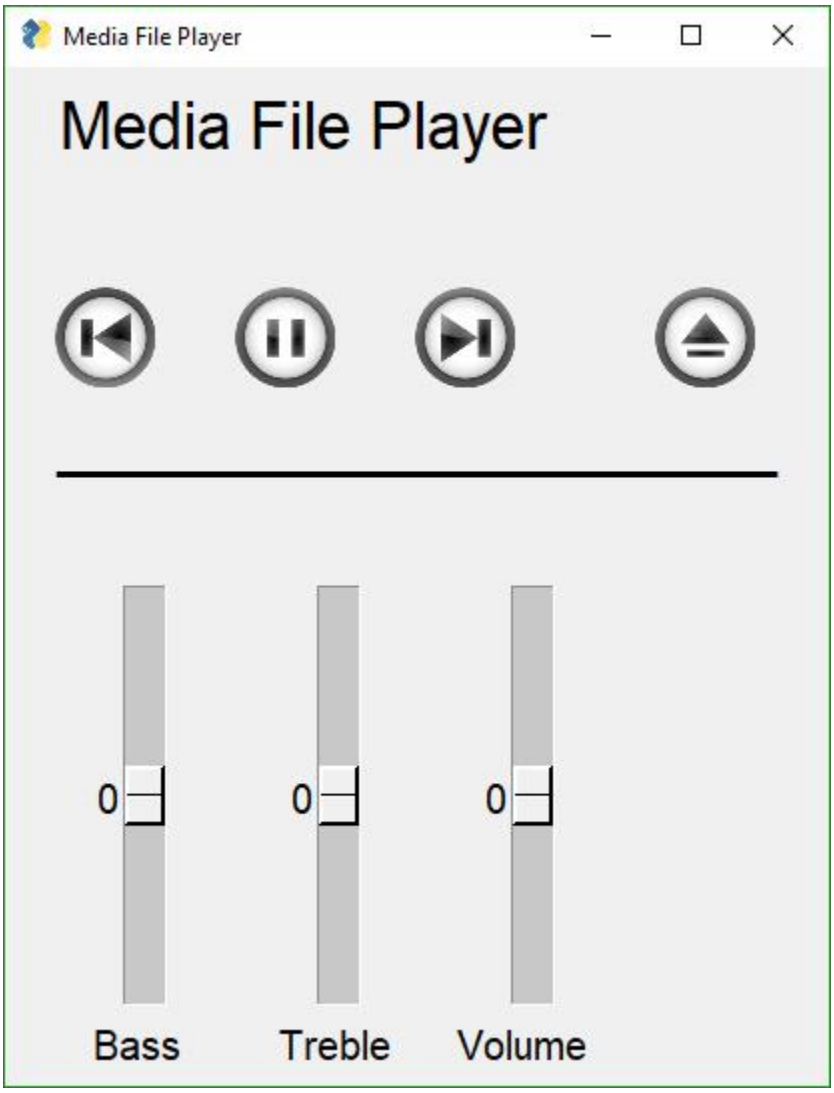


In addition to a primary GUI, you can add a Progress Meter to your code with ONE LINE of code. Slide this line into any of your `for` loops and get a nice meter:

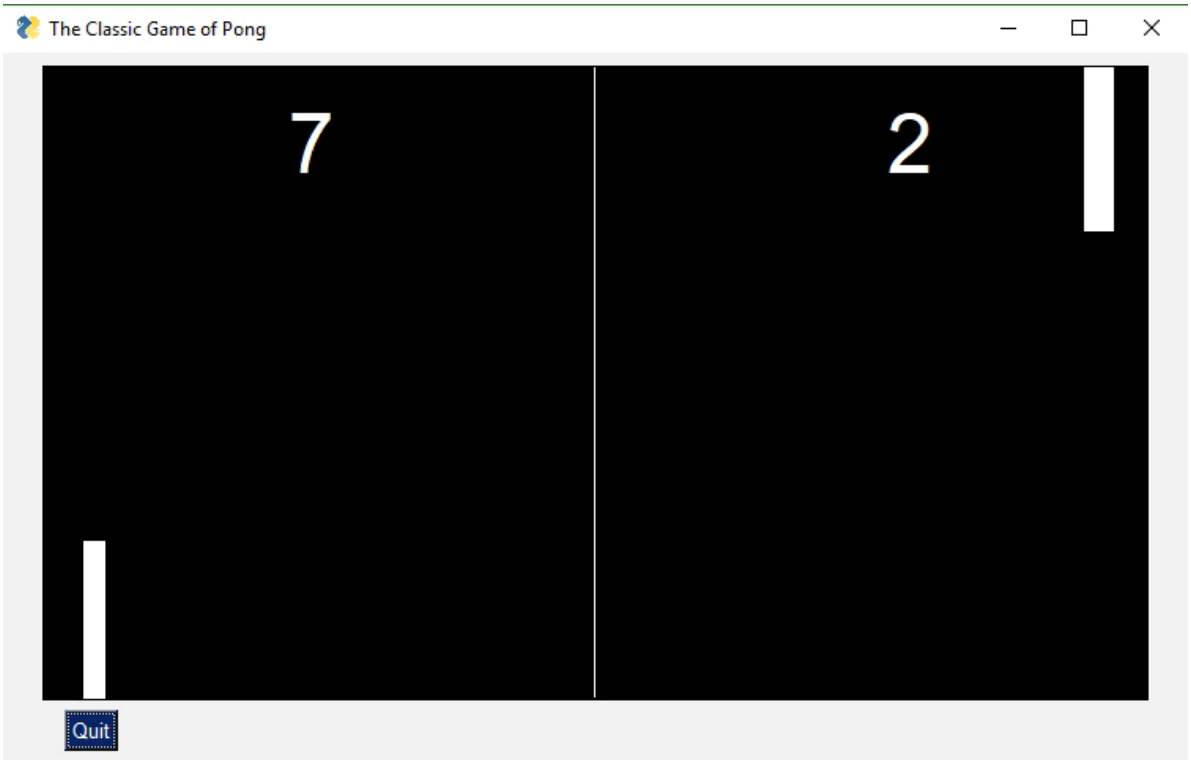
```
OneLineProgressMeter('My meter title', current_value, max value, 'key')
```



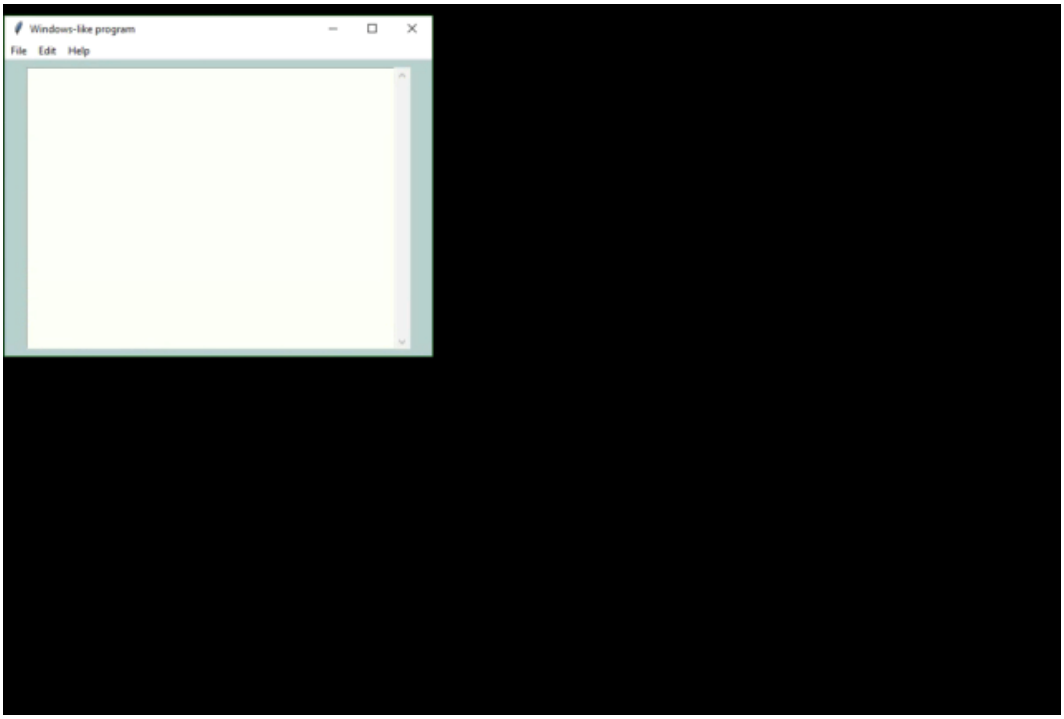
You can build an async media player GUI with custom buttons in 30 lines of code.



How about embedding a game inside of a GUI? This game of Pong is written in tkinter and then dropped into the PySimpleGUI window creating a game that has an accompanying GUI.



Combining PySimpleGUI with PyInstaller creates something truly remarkable and special, a Python program that looks like a Windows WinForms application. This application with working menu was created in 20 lines of Python code. It is a single .EXE file that launches straight into the screen you see. And more good news, the only icon you see on the taskbar is the window itself... there is no pesky shell window.



Background

I was frustrated by having to deal with the dos prompt when I had a powerful Windows machine right in front of me. Why is it SO difficult to do even the simplest of input/output to a window in Python??

There are a number of 'easy to use' Python GUIs, but they were too limited for my requirements. PySimpleGUI aims for the same simplicity found in packages like `EasyGUI`and `WxSimpleGUI` , both really handy but limited, and adds the ability to define your own layouts. This ability to make your own windows using a large palette of widgets is but one difference between the existing "simple" packages and `PySimpleGUI`.
With a simple GUI, it becomes practical to "associate" .py files with the python interpreter on Windows. Double click a py file and up pops a GUI window, a more pleasant experience than opening a dos Window and typing a command line.

The `PySimpleGUI` package is focused on the ***developer***.
Create a custom GUI with as little and as simple code as possible.

This was the primary focus used to create PySimpleGUI.

"Do it in a Python-like way"

was the second.

Features

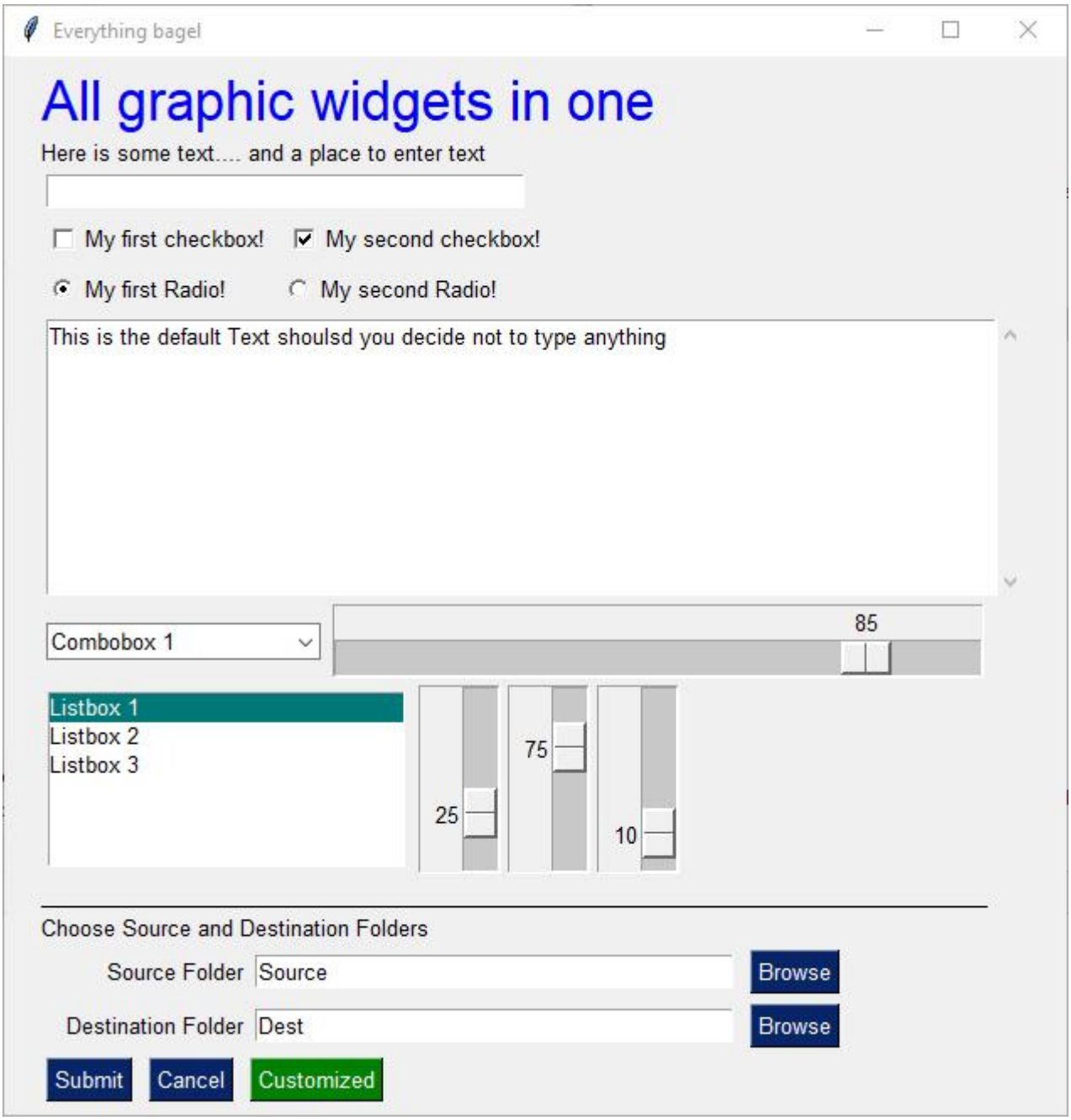
While simple to use, PySimpleGUI has significant depth to be explored by more advanced programmers. The feature set goes way beyond the requirements of a beginner programmer, and into the required features needed for complex GUIs.

```
Features of PySimpleGUI include:
    Support for versions Python 2.7 and 3
    Text
    Single Line Input
    Buttons including these types:
        File Browse
        Files Browse
        Folder Browse
        SaveAs
        Non-closing return
        Close window
        Realtime
        Calendar chooser
        Color chooser
    Checkboxes
    Radio Buttons
    Listbox
    Option Menu
    Slider
    Graph
    Frame with title
    Icons
```

```
    Multi-line Text Input
    Scroll-able Output
    Images
    Progress Bar            Async/Non-Blocking Windows
    Tabbed windows
    Persistent Windows
    Redirect Python Output/Errors to scrolling window
    'Higher level' APIs (e.g. MessageBox, YesNobox, ...)
    Single-Line-Of-Code Proress Bar & Debug Print
    Complete control of colors, look and feel
    Selection of pre-defined palettes
    Button images
    Return values as dictionary
    Set focus
    Bind return key to buttons
    Group widgets into a column and place into window anywhere
    Scrollable columns
    Keyboard low-level key capture
    Mouse scroll-wheel support
    Get Listbox values as they are selected
    Get slider, spinner, combo as they are changed
    Update elements in a live window
    Bulk window-fill operation
    Save / Load window to/from disk
    Borderless (no titlebar) windows
    Always on top windows
    Menus with ALT-hotkey
    Tooltips
    Clickable links
    No async programming required (no callbacks to worry about)
```

An example of many widgets used on a single window. A little further down you'll find the 21 lines of code required to create this complex window. Try it if you don't believe it. Install PySimpleGUI then :

Start Python, copy and paste the code below into the >>> prompt and hit enter. This will pop up...




```
import PySimpleGUI as sg

layout = [[sg.Text('All graphic widgets in one window!', size=(30, 1),
font=("Helvetica", 25), text_color='blue')],
   [sg.Text('Here is some text.... and a place to enter text')],
   [sg.InputText()],
   [sg.Checkbox('My first checkbox!'), sg.Checkbox('My second checkbox!',
default=True)],
   [sg.Radio('My first Radio!     ', "RADIO1", default=True), sg.Radio('My second
Radio!', "RADIO1")],
   [sg.Multiline(default_text='This is the default Text shoulsd you decide not to
type anything',)],
[sg.InputCombo(['Combobox 1', 'Combobox 2'], size=(20, 3)),
 sg.Slider(range=(1, 100), orientation='h', size=(35, 20), default_value=85)],
[sg.Listbox(values=['Listbox 1', 'Listbox 2', 'Listbox 3'], size=(30, 6)),
```

```
 sg.Slider(range=(1, 100), orientation='v', size=(10, 20), default_value=25),
 sg.Slider(range=(1, 100), orientation='v', size=(10, 20), default_value=75),
 sg.Slider(range=(1, 100), orientation='v', size=(10, 20), default_value=10)],
[sg.Text('_'  * 100, size=(70, 1))],
[sg.Text('Choose Source and Destination Folders', size=(35, 1))],
[sg.Text('Source Folder', size=(15, 1), auto_size_text=False, justification='right'),
sg.InputText('Source'),
 sg.FolderBrowse()],
[sg.Text('Destination Folder', size=(15, 1), auto_size_text=False,
justification='right'), sg.InputText('Dest'),
 sg.FolderBrowse()],
[sg.Submit(), sg.Cancel(), sg.Button('Customized', button_color=('white', 'green'))]]

button, values  = sg.Window('Everything bagel', auto_size_text=True,
default_element_size=(40, 1)).LayoutAndRead(layout)
```

Design Goals

Copy, Paste, Run.

`PySimpleGUI's` goal with the API is to be easy on the programmer, and to function in a Python-like way. Since GUIs are visual, it was desirable for the code to visually match what's on the screen. By providing a significant amount of documentation and an easy to use Cookbook, it's possible to see your first GUI within 5 minutes of beginning the installation.

Be Pythonic

Be Pythonic... Attempted to use language constructs in a natural way and to exploit some of Python's interesting features. Python's lists and optional parameters make PySimpleGUI work smoothly.

- windows are represented as Python lists.
    - A window is a list of rows
    - A row is a list of elements
- Return values are a list of button presses and input values.
- Return values can also be represented as a dictionary
- The SDK calls collapse down into a single line of Python code that presents a custom GUI and returns values
- Linear programming instead of callbacks

Lofty Goals

Change Python

The hope is not that ***this*** package will become part of the Python Standard Library.

The hope is that Python will become ***the*** go-to language for creating GUI programs that run on Windows, Mac, and Linux *for all levels of developer*.

The hope is that beginners that are interested in graphic design will have an easy way to express themselves, right from the start of their Python experience.

There is a noticeable gap in the Python GUI solution. Fill that gap and who knows what will happen.

Maybe there's no "there there". ***Or*** maybe a simple GUI API will enable Python to dominate yet another computing discipline like it has so many others. This is my attempt to find out.

Getting Started with PySimpleGUI

Installing Python 3

```
pip install --upgrade PySimpleGUI
```
On some systems you need to run pip3.

```
pip3 install --upgrade PySimpleGUI
```
On a Raspberry Pi, this is should work:

```
sudo pip3 install --upgrade pysimplegui
```
Some users have found that upgrading required using an extra flag on the pip `--no-cache-dir`.
```
pip install --upgrade --no-cache-dir
```
On some versions of Linux you will need to first install pip. Need the Chicken before you can get the Egg (get it... Egg?)

```
sudo apt install python3-pip
```
If for some reason you are unable to install using `pip`, don't worry, you can still import PySimpleGUI by downloading the file PySimleGUI.py and placing it in your folder along with the application that is importing it.
`tkinter` is a requirement for PySimpleGUI (the only requirement). Some OS variants, such as Ubuntu, do not some with `tkinter` already installed. If you get an error similar to:
```
ImportError: No module named tkinter
```
then yosudou need to install `tkinter`. Be sure and get the Python 3 version. ` `sudo apt-get install python3-tk`

Installing for Python 2.7

```
pip install --upgrade PySimpleGUI27
```
Python 2.7 support is relatively new and the bugs are still being worked out. I'm unsure what may need to be done to install tkinter for Python 2.7. Will update this readme when more info is available

Like above, you may have to install either pip or tkinter. To do this on Python 2.7:

```
sudo apt install python-pip
sudo apt install python-tkinter
```

Testing your installation

Once you have installed, or copied the .py file to your app folder, you can test the installation using python. At the command prompt start up Python.

Instructions for Python 2.7:

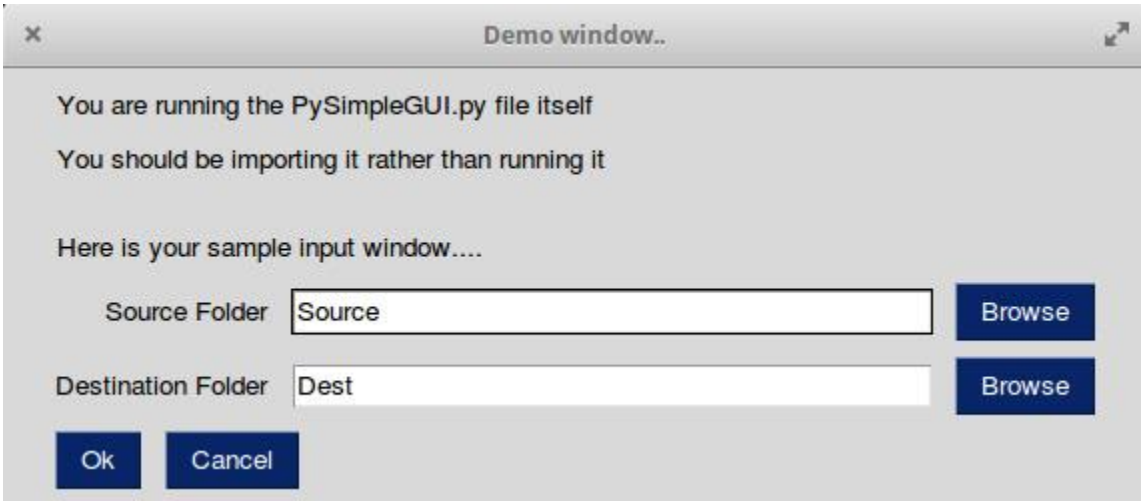
```
python
>>> import PySimpleGUI27
>>> PySimpleGUI27.main()
```

Instructions for Python 3:

```
python3
>>> import PySimpleGUI
>>> PySimpleGUI.main()
```

You will see a sample window in the center of your screen. If it's not installed correctly you are likely to get an error message during one of those commands

Here is the window you should see:



Prerequisites

Python 2.7 or Python 3 tkinter

PySimpleGUI Runs on all Python3 platforms that have tkinter running on them. It has been tested on Windows, Mac, Linux, Raspberry Pi. Even runs on `pypy3`.

EXE file creation

If you wish to create an EXE from your PySimpleGUI application, you will need to install `PyInstaller`. There are instructions on how to create an EXE at the bottom of this ReadMe

Using - Python 3

To use in your code, simply import.... `import PySimpleGUI as sg`
Then use either "high level" API calls or build your own windows.

```
sg.Popup('This is my first Popup')
```

Yes, it's just that easy to have a window appear on the screen using Python. With PySimpleGUI, making a custom window appear isn't much more difficult. The goal is to get you running on your GUI within ***minutes***, not hours nor days.

Using - Python 2.7

Those using Python 2.7 will import a different module name `import PySimpleGUI27 as sg`

Code Samples Assume Python 3

While all of the code examples you will see in this Readme and the Cookbook assume Python 3 and thus have an `import PySimpleGUI` at the top, you can run ***all*** of this code on Python 2.7 by changing the import statement to `import PySimpleGUI27`

APIs

PySimpleGUI can be broken down into 2 types of API's:

- High Level single call functions (The `Popup` calls)
- Custom window functions

Python Language Features

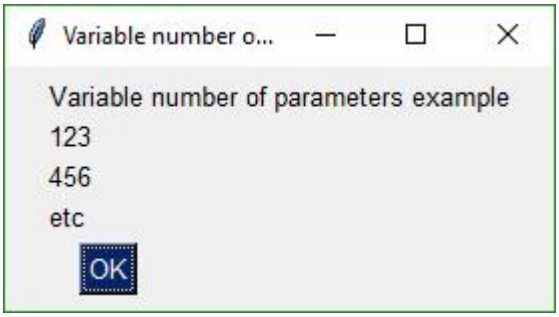
There are a number of Python language features that PySimpleGUI utilizes heavily for API access that should be understood...

- Variable number of arguments to a function call
- Optional parameters to a function call
- Dictionaries

Variable Number of Arguments

The "High Level" API calls that *output* values take a variable number of arguments so that they match a "print" statement as much as possible. The idea is to make it simple for the programmer to output as many items as desired and in any format. The user need not convert the variables to be output into the strings. The PySimpleGUI functions do that for the user.

```
sg.Popup('Variable number of parameters example', var1, var2, "etc")
```
Each new item begins on a new line in the Popup

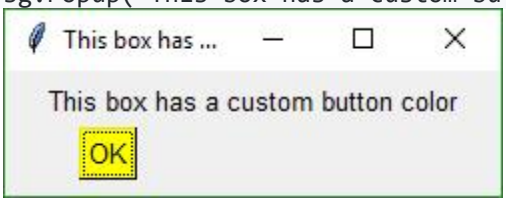


Optional Parameters to a Function Call

This feature of the Python language is utilized ***heavily*** as a method of customizing windows and window Elements. Rather than requiring the programmer to specify every possible option for a widget, instead only the options the caller wants to override are specified.

Here is the function definition for the Popup function. The details aren't important. What is important is seeing that there is a long list of potential tweaks that a caller can make. However, they don't *have* to be specified on each and every call.

```
def Popup(*args,
          button_color=None,
          button_type=MSG_BOX_OK,
          auto_close=False,
          auto_close_duration=None,
          icon=DEFAULT_WINDOW_ICON,
          line_width=MESSAGE_BOX_LINE_WIDTH,
          font=None):
```
If the caller wanted to change the button color to be black on yellow, the call would look something like this:

```
sg.Popup('This box has a custom button color', button_color=('black', 'yellow'))
```


Dictionaries

Dictionaries are used by more advanced PySimpleGUI users. You'll know that dictionaries are being used if you see a keyparameter on any Element. Dictionaries are used in 2 ways:

1. To identify values when a window is read
2. To identify Elements so that they can be "updated"

High Level API Calls - Popup's

"High level calls" are those that start with "Popup". They are the most basic form of communications with the user. They are named after the type of window they create, a pop-up window. These windows are meant to be short lived while, either delivering information or collecting it, and then quickly disappearing.

Popup Output

Think of the `Popup` call as the GUI equivalent of a `print` statement. It's your way of displaying results to a user in the windowed world. Each call to Popup will create a new Popup window.
`Popup` calls are normally blocking. your program will stop executing until the user has closed the Popup window. A non-blocking window of Popup discussed in the async section.
Just like a print statement, you can pass any number of arguments you wish. They will all be turned into strings and displayed in the popup window.

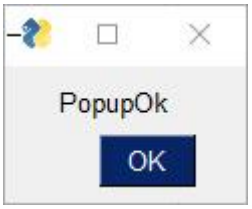
There are a number of Popup output calls, each with a slightly different look (e.g. different button labels).

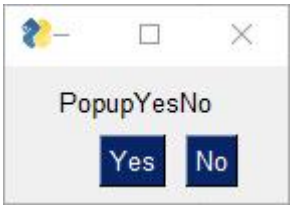
The list of Popup output functions are

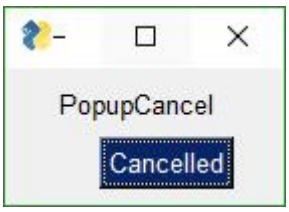
```
Popup
PopupOk
PopupYesNo
PopupCancel
PopupOkCancel
PopupError
PopupTimed, PopupAutoClose
PopupNoWait, PopupNonBlocking
```

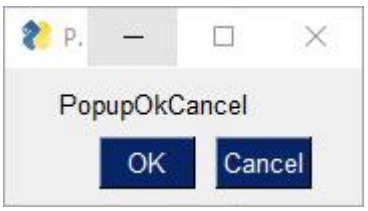
The trailing portion of the function name after Popup indicates what buttons are shown. `PopupYesNo` shows a pair of button with Yes and No on them. `PopupCancel` has a Cancel button, etc.
While these are "output" windows, they do collect input in the form of buttons. The Popup functions return the button that was clicked. If the Ok button was clicked, then Popup returns the string 'Ok'. If the user clicked the X button to close the window, then the button value returned is `None`.
The function `PopupTimed` or `PopupAutoClose` are popup windows that will automatically close after come period of time.
Here is a quick-reference showing how the Popup calls look.

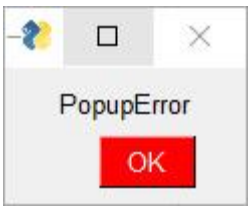
```
sg.Popup('Popup')
sg.PopupOk('PopupOk')
sg.PopupYesNo('PopupYesNo')
sg.PopupCancel('PopupCancel')
sg.PopupOkCancel('PopupOkCancel')
sg.PopupError('PopupError')
sg.PopupTimed('PopupTimed')
sg.PopupAutoClose('PopupAutoClose')
```

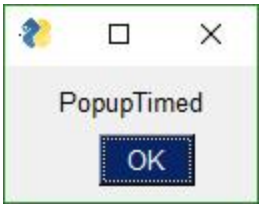








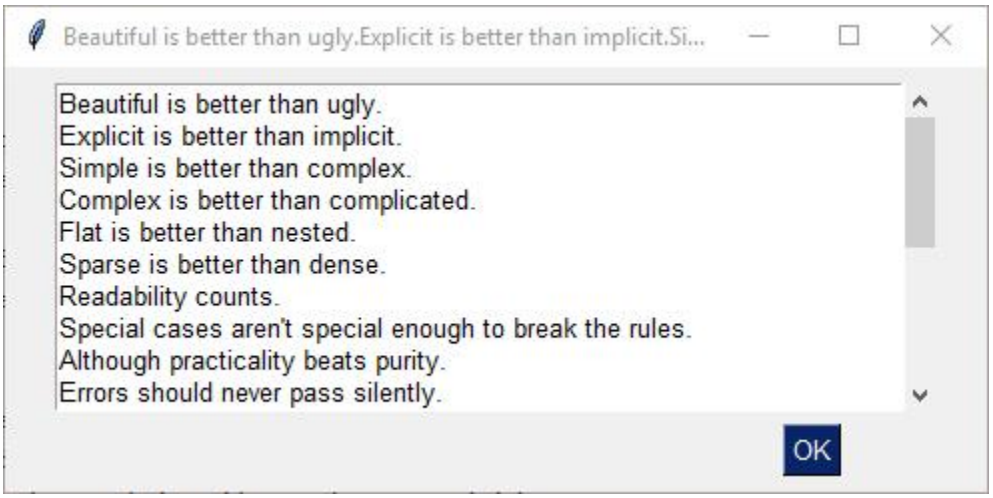




Scrolled Output

There is a scrolled version of Popups should you have a lot of information to display.

```
sg.PopupScrolled(my_text)
```



The `PopupScrolled` will auto-fit the window size to the size of the text. Specify `None` in the height field of a `size` parameter to get auto-sized height.
This call will create a scrolled box 80 characters wide and a height dependent upon the number of lines of text.

```
sg.PopupScrolled(my_text, size=(80, None))
```

Note that the default max number of lines before scrolling happens is set to 50. At 50 lines the scrolling will begin.

PopupNoWait

The Popup call PopupNoWait or PopupNonBlocking will create a popup window and then immediately return control back to you. All other popup functions will block, waiting for the user to close the popup window.

This function is very handy for when you're **debugging** and want to display something as output but don't want to change the programs's overall timing by blocking. Think of it like a `print` statement
A word of ***caution***... Windows that are created after the NoWait Popup are "slaves" to the NoWait'd popup. If you close the Popup, it will also lose the window the you created after the Popup. A good rule of thumb is to leave the popup open while you're interacting with the rest of your program until you understand what happens when you close the NoWaitPopup.

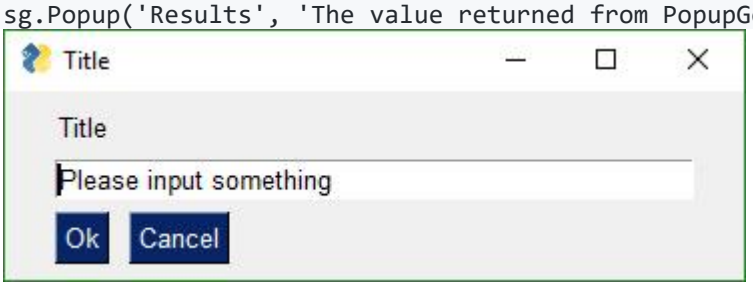
Popup Input

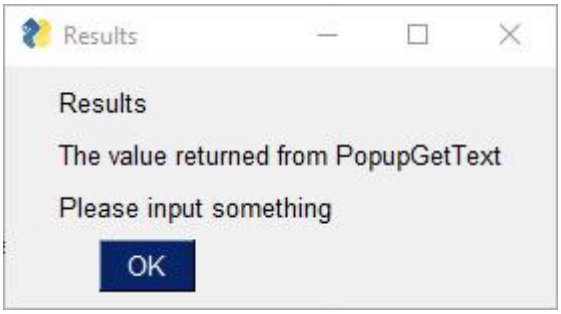
There are Popup calls for single-item inputs. These follow the pattern of `Popup` followed by `Get` and then the type of item to get.

- `PopupGetString` - get a single line of text
- `PopupGetFile` - get a filename
- `PopupGetFolder` - get a folder name

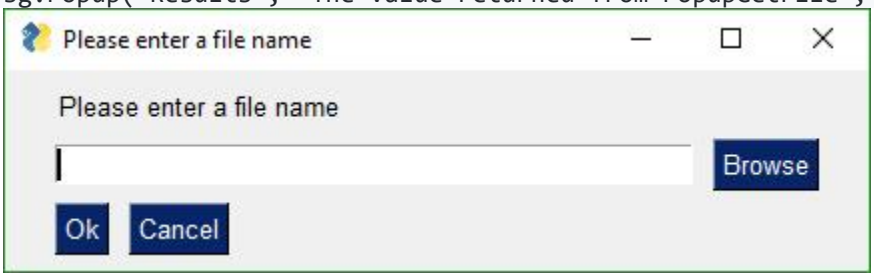
Rather than make a custom window to get one data value, call the Popup input function to get the item from the user.

```
import PySimpleGUI as sg

text = sg.PopupGetText('Title', 'Please input something')
sg.Popup('Results', 'The value returned from PopupGetText', text)
```



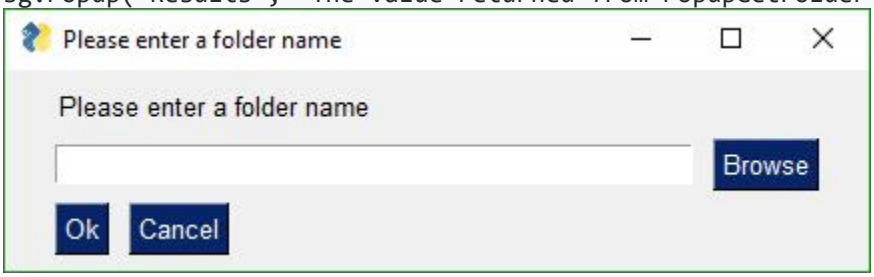



```
text = sg.PopupGetFile('Please enter a file name')
sg.Popup('Results', 'The value returned from PopupGetFile', text)
```



The window created to get a folder name looks the same as the get a file name. The difference is in what the browse button does. `PopupGetFile` shows an Open File dialog box while `PopupGetFolder` shows an Open Folder dialog box.

```
text = sg.PopupGetFolder('Please enter a folder name')
sg.Popup('Results', 'The value returned from PopupGetFolder', text)
```



Progress Meters!

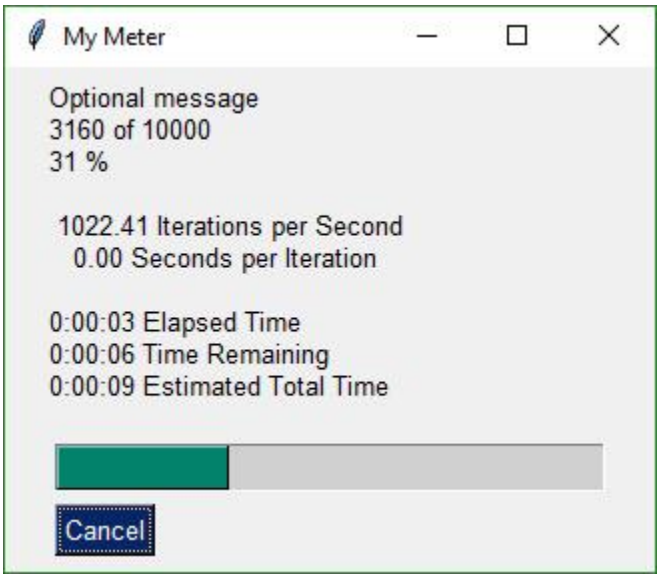
We all have loops in our code. 'Isn't it joyful waiting, watching a counter scrolling past in a text window? How about one line of code to get a progress meter, that contains statistics about your code?

```
OneLineProgressMeter(title,
                     current_value,
                     max_value,
                     key,
                     *args,
                     orientation=None,
                     bar_color=DEFAULT_PROGRESS_BAR_COLOR,
                     button_color=None,
                     size=DEFAULT_PROGRESS_BAR_SIZE,
                     border_width=DEFAULT_PROGRESS_BAR_BORDER_WIDTH):
```

Here's the one-line Progress Meter in action!

```
for i in range(1,10000):
    sg.OneLineProgressMeter('My Meter', i+1, 10000, 'key','Optional message')
```

That line of code resulted in this window popping up and updating.



A meter AND fun statistics to watch while your machine grinds away, all for the price of 1 line of code. With a little trickery you can provide a way to break out of your loop using the Progress Meter window. The cancel button results in a `False` return value from `OneLineProgressMeter`. It normally returns `True`.

Be sure and add one to your loop counter so that your counter goes from 1 to the max value. If you do not add one, your counter will never hit the max value. Instead it will go from 0 to max-1.

Debug Output

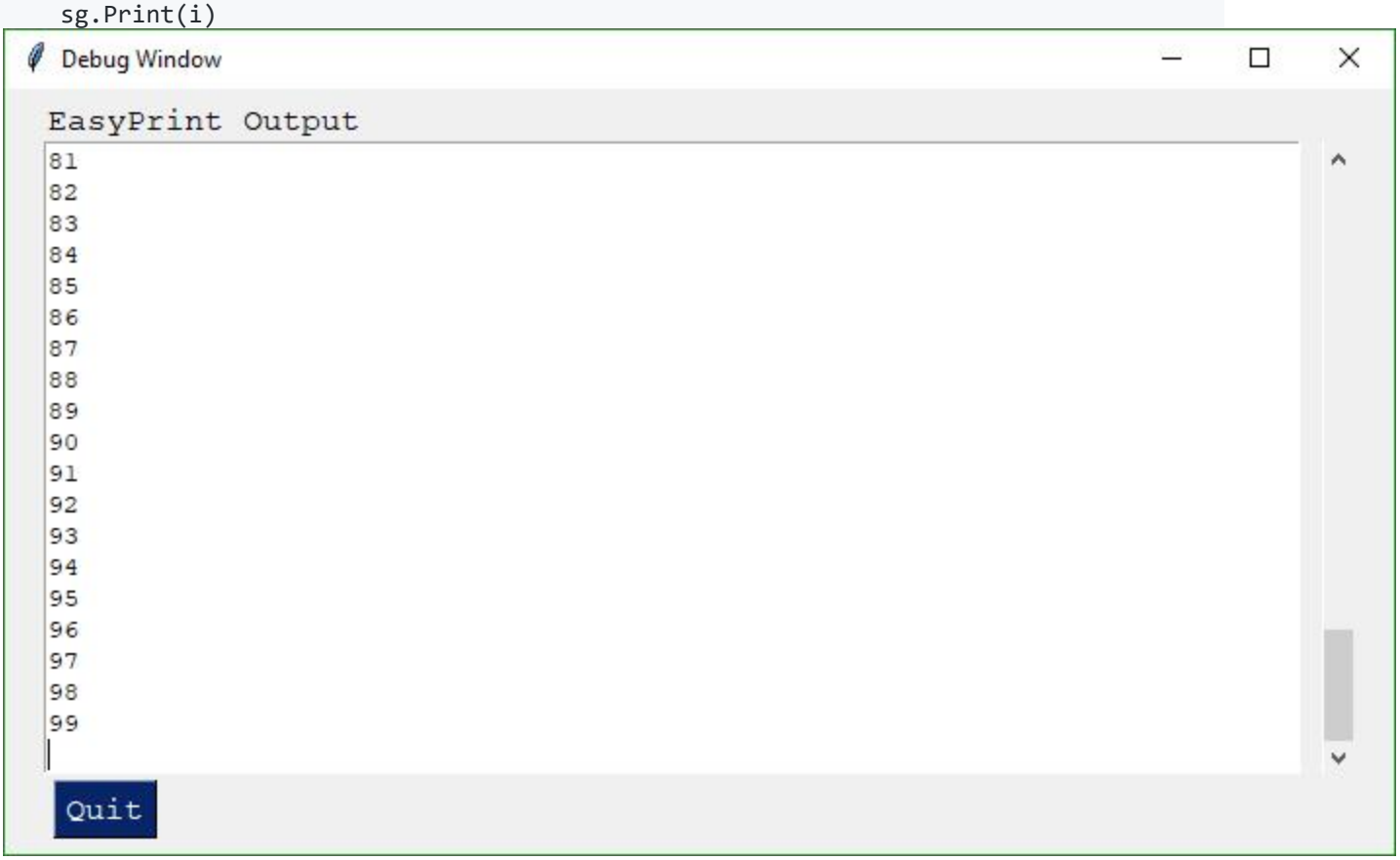
Another call in the 'Easy' families of APIs is `EasyPrint`. It will output to a debug window. If the debug window isn't open, then the first call will open it. No need to do anything but stick a 'print' call in your code. You can even replace your 'print' calls with calls to EasyPrint by simply sticking the statement

```
print = sg.EasyPrint
```

at the top of your code. There are a number of names for the same EasyPrint function. `Print` is one of the better ones to use as it's easy to remember. It is simply `print` with a capital P.

```
import PySimpleGUI as sg

for i in range(100):
```

```
    sg.Print(i)
```



Or if you didn't want to change your code:

```
import PySimpleGUI as sg

print=sg.Print
for i in range(100):
    print(i)
```

Just like the standard print call, `EasyPrint` supports the `sep` and `end` keyword arguments. Other names that can be used to call `EasyPrint` include Print, `eprint`, If you want to close the window, call the function `EasyPrintClose`.
A word of caution. There are known problems when multiple PySimpleGUI windows are opened, particularly if the user closes them in an unusual way. Not a reason to stay away from using it. Just something to keep in mind if you encounter a problem.

You can change the size of the debug window using the `SetOptions` call with the `debug_win_size` parameter.

Custom window API Calls (Your First window)

This is the FUN part of the programming of this GUI. In order to really get the most out of the API, you should be using an IDE that supports auto complete or will show you the definition of the function. This will make customizing go smoother.

This first section on custom windows is for your typical, blocking, non-persistant window. By this I mean, when you "show" the window, the function will not return until the user has clicked a button or closed the window. When this happens, the window will be automatically closed.

Two other types of windows exist.

1. Persistent window - rather than closing on button clicks, the show window function returns and the window continues to be visible. This is good for applications like a chat window.
2. Asynchronous window - the trickiest of the lot. Great care must be exercised. Examples are an MP3 player or status dashboard. Async windows are updated (refreshed) on a periodic basis.

It's both not enjoyable nor helpful to immediately jump into tweaking each and every little thing available to you.

The window Designer

The good news to newcomers to GUI programming is that PySimpleGUI has a window designer. Better yet, the window designer requires no training and everyone knows how to use it.

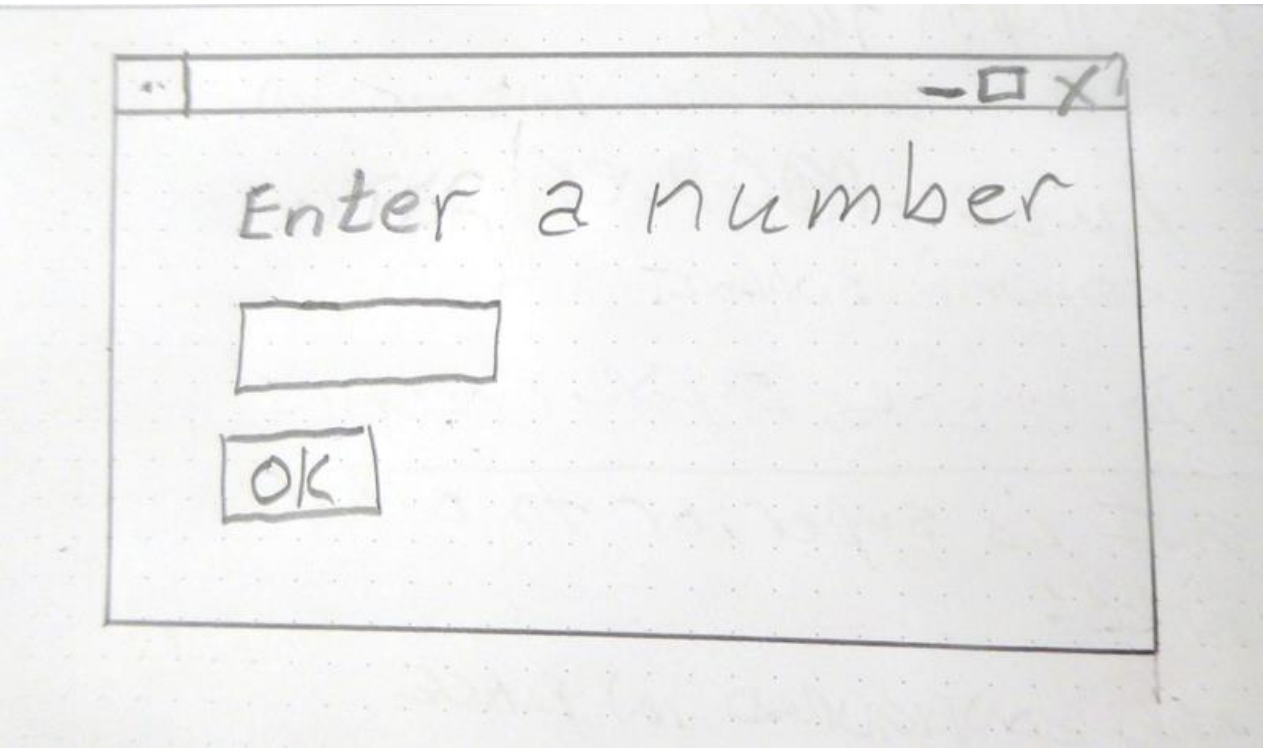
It's a manual process, but if you follow the instructions, it will take only a minute to do and the result will be a nice looking GUI. The steps you'll take are:

1. Sketch your GUI on paper
2. Divide your GUI up into rows
3. Label each Element with the Element name
4. Write your Python code using the labels as pseudo-code

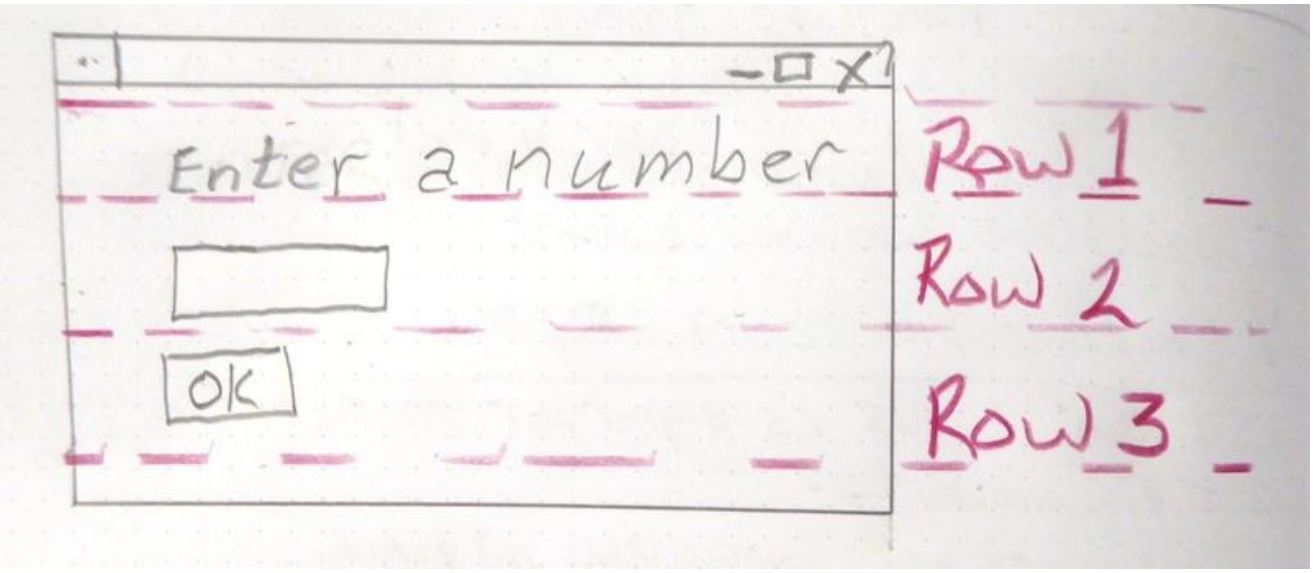
Let's take a couple of examples.

Enter a number.... Popular beginner programs are often based on a game or logic puzzle that requires the user to enter something, like a number. The "high-low" answer game comes to mind where you try to guess the number based on high or low tips.

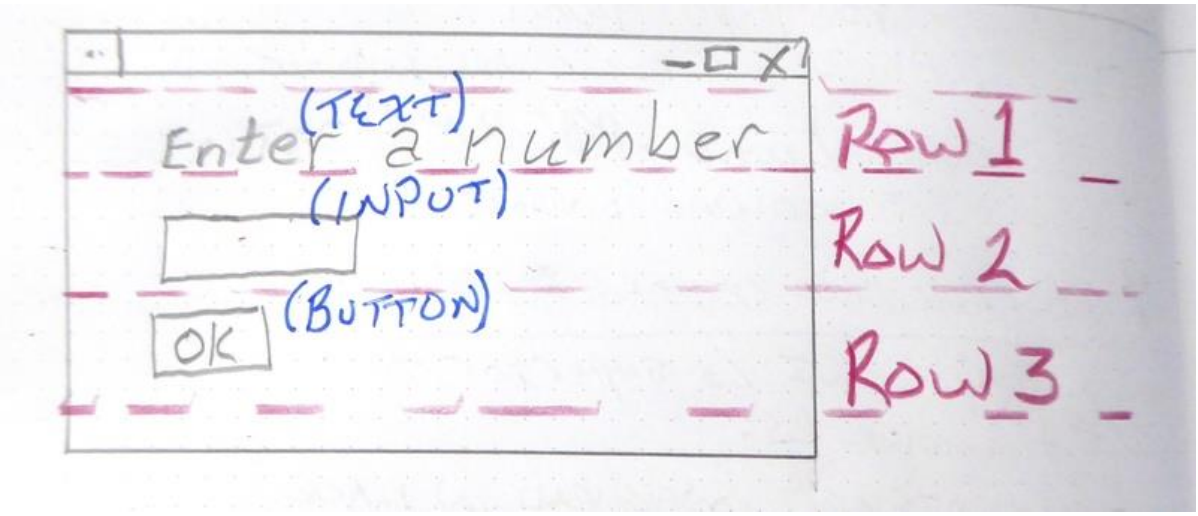
Step 1- Sketch the GUI



Step 2 - Divide into rows



Step 3 - Label elements



Step 4 - Write the code The code we're writing is the layout of the GUI itself. This tutorial only focuses on getting the window code written, not the stuff to display it, get results.

We have only 1 element on the first row, some text. Rows are written as a "list of elements", so we'll need [ ] to make a list. Here's the code for row 1

```
[ sg.Text('Enter a number') ]
```

Row 2 has 1 elements, an input field.

```
[ sg.Input() ]
```

Row 3 has an OK button

```
[ sg.OK() ]
```

Now that we've got the 3 rows defined, they are put into a list that represents the entire window.

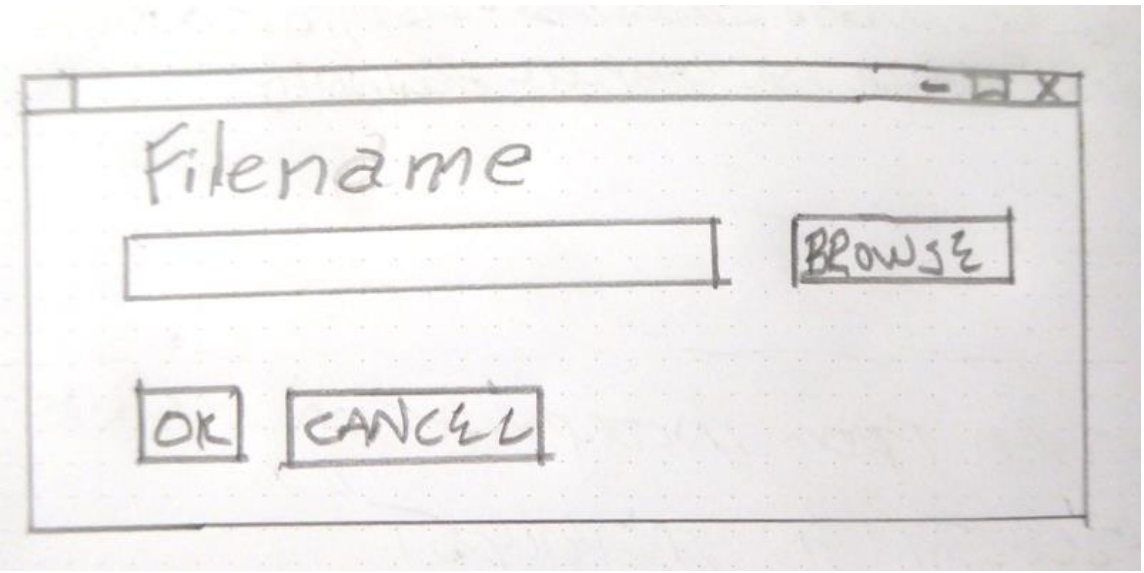
```
layout = [ [sg.Text('Enter a Number')],
           [sg.Input()],
           [sg.OK()] ]
```

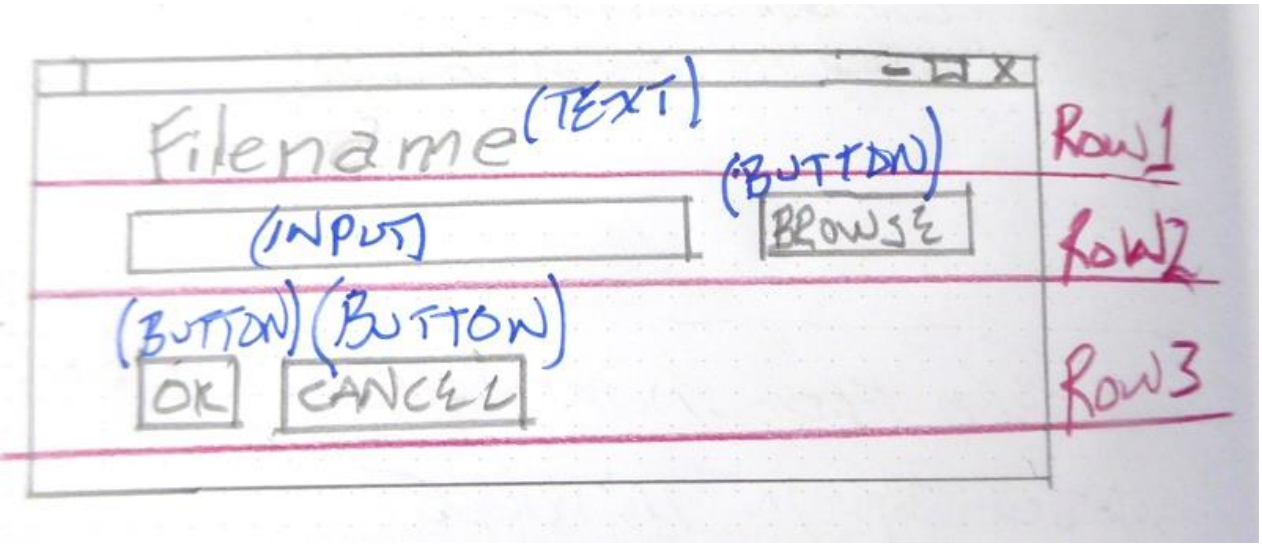
Finally we can put it all together into a program that will display our window.

```
import PySimpleGUI as sg

layout = [[sg.Text('Enter a Number')],
          [sg.Input()],
          [sg.OK()] ]

button, (number,) = sg.Window('Enter a number example').LayoutAndRead(layout)

sg.Popup(button, number)
```

Example 2 - Get a filename

Let's say you've got a utility you've written that operates on some input file and you're ready to use a GUI to enter than filename rather than the command line. Follow the same steps as the previous example - draw your window on paper, break it up into rows, label the elements.





Writing the code for this one is just as straightforward. There is one tricky thing, that browse for a file button. Thankfully PySimpleGUI takes care of associating it with the input field next to it. As a result, the code looks almost exactly like the window on the paper.

```
import PySimpleGUI as sg

layout = [[sg.Text('Filename')],
          [sg.Input(), sg.FileBrowse()],
          [sg.OK(), sg.Cancel()] ]

button, (number,) = sg.Window('Get filename example').LayoutAndRead(layout)

sg.Popup(button, number)
```

Read on for detailed instructions on the calls that show the window and return your results.

Copy these design patterns!

All of your PySimpleGUI programs will utilize one of these 3 design patterns depending on the type of window you're implementing.

Pattern 1 - Single read windows

This is the most basic design pattern. Use this for windows that are shown to the user 1 time. The input values are gathered and returned to the program

```
window_rows = [[sg.Text('SHA-1 and SHA-256 Hashes for the file')],
               [sg.InputText(), sg.FileBrowse()],
               [sg.Submit(), sg.Cancel()]]

window = sg.Window('SHA-1 & 256 Hash')

button, (source_filename,) = window.LayoutAndRead(window_rows)
```

Pattern 2 - Single-read window "chained"

Python has a ***beautiful*** way of compacting code known as "chaining". You take the output from one function and feed it as input to the next. Notice in the first example how a window is first obtained by calling Window and then that window is then read. It's possible to combine the creation of the window with the read. This design pattern does exactly that, chain together the window creation and the window reading.

```
window_rows = [[sg.Text('SHA-1 and SHA-256 Hashes for the file')],
               [sg.InputText(), sg.FileBrowse()],
               [sg.Submit(), sg.Cancel()]]

button, (source_filename,) = sg.Window('SHA-1 & 256 Hash').LayoutAndRead(window_rows)
```

Pattern 3 - Persistent window (multiple reads)

Some of the more advanced 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 output information to the user and gather input data.

This is done by splitting the LayoutAndRead call apart into a Layout call and a Read call. Note how chaining is again used. In this case a window is created by calling Window which is then passed on to the Layout method. The Layout method returns the window value so that it can be stored and used later in the program to Read the window.

```
import PySimpleGUI as sg

layout = [[sg.Text('Persistent window')],
          [sg.RButton('Turn LED On')],
          [sg.RButton('Turn LED Off')],
          [sg.Exit()]]
```

```
window = sg.Window('Raspberry Pi GUI').Layout(layout)

while True:
    button, values = window.Read()
    if button is None:
        break
```

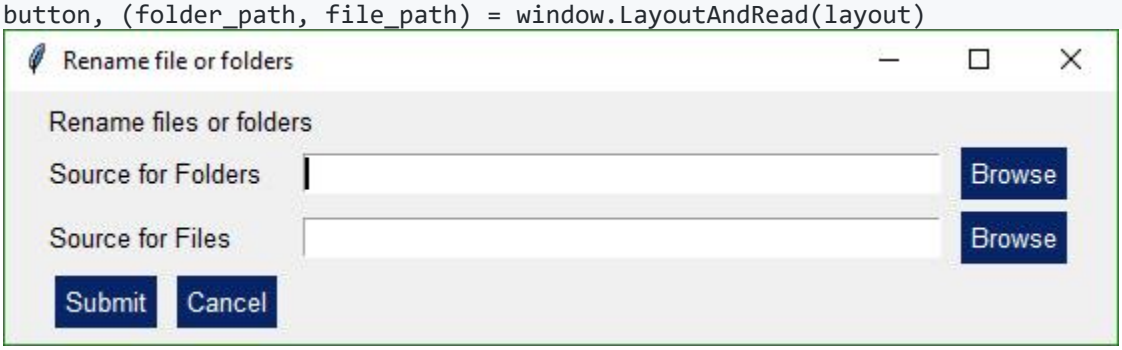
How GUI Programming in Python Should Look? At least for beginners

Why is Python such a great teaching language and yet no GUI framework exists that lends itself to the basic building blocks of Python, the list or dictionary? PySimpleGUI set out to be a Pythonic solution to the GUI problem. Whether it achieved this goal is debatable, but it was an attempt just the same.

The key to custom windows in PySimpleGUI is to view windows as ROWS of Elements. Each row is specified as a list of these Elements. Put the rows together and you've got a window.

Let's dissect this little program

```
import PySimpleGUI as sg

layout = [[sg.Text('Rename files or folders')],
          [sg.Text('Source for Folders', size=(15, 1)), sg.InputText(),
sg.FolderBrowse()],
          [sg.Text('Source for Files ', size=(15, 1)), sg.InputText(),
sg.FolderBrowse()],
          [sg.Submit(), sg.Cancel()]]

window = sg.Window('Rename Files or Folders')

button, (folder_path, file_path) = window.LayoutAndRead(layout)
```



Let's agree the window has 4 rows.

The first row only has **text** that reads `Rename files or folders`
The second row has 3 elements in it. First the **text** `Source for Folders`, then an **input** field, then a **browse** button.
Now let's look at how those 2 rows and the other two row from Python code:

```
layout = [[sg.Text('Rename files or folders')],
          [sg.Text('Source for Folders', size=(15, 1)), sg.InputText(),
sg.FolderBrowse()],
```

```
          [sg.Text('Source for Files ', size=(15, 1)), sg.InputText(),
sg.FolderBrowse()],
          [sg.Submit(), sg.Cancel()]]
```
See how the source code mirrors the layout? You simply make lists for each row, then submit that table to PySimpleGUI to show and get values from.

And what about those return values? Most people simply want to show a window, get the input values and do something with them. So why break up the code into button callbacks, etc, when I simply want my window's input values to be given to me.

For return values the window is scanned from top to bottom, left to right. Each field that's an input field will occupy a spot in the return values.

In our example window, there are 2 fields, so the return values from this window will be a list with 2 values in it.

```
button, (folder_path, file_path) = window.LayoutAndRead(layout)
```
In the statement that shows and reads the window, the two input fields are directly assigned to the caller's variables `folder_path` and `file_path`, ready to use. No parsing no callbacks.
Isn't this what almost every Python programmer looking for a GUI wants?? Something easy to work with to get the values and move on to the rest of the program, where the real action is taking place. Why write pages of GUI code when the same layout can be achieved with PySimpleGUI in 3 or 4 lines of code. 4 lines or 40? I chose 4.

Return values

As of version 2.8 there are 2 forms of return values, list and dictionary.

Return values as a list

By default return values are a list of values, one entry for each input field.

Return information from Window, SG's primary window builder interface, is in this format:

```
button, (value1, value2, ...)
```
Each of the Elements that are Input Elements will have a value in the list of return values. You can unpack your GUI directly into the variables you want to use.

```
button, (filename, folder1, folder2, should_overwrite) =
window.LayoutAndRead(window_rows)
```
Or, you can unpack the return results separately.

```
button, values = window.LayoutAndRead(window_rows)
filename, folder1, folder2, should_overwrite = values
```
If you have a SINGLE value being returned, it is written this way:

```
button, (value1,) = window.LayoutAndRead(window_rows)
```
Another way of parsing the return values is to store the list of values into a variable representing the list of values and then index each individual value. This is not the preferred way of doing it.

```
button, value_list = window.LayoutAndRead(window_rows)
value1 = value_list[0]
value2 = value_list[1]
...
```

Return values as a dictionary

For windows longer than 3 or 4 fields you will want to use a dictionary to help you organize your return values. In almost all (if not all) of the demo programs you'll find the return values being passed as a dictionary. It is not a difficult concept to grasp, the syntax is easy to understand, and it makes for very readable code.

The most common window read statement you'll encounter looks something like this:

```
button, values = window.LayoutAndRead(layout)
```
or

```
button, values = window.Read()
```
All of your return values will be stored in the variable `values`. When using the dictionary return values, the `values` variable is a dictionary.
To use a dictionary, you will need to:

- Mark each input element you wish to be in the dictionary with the keyword `key`.

If **any** element in the window has a `key`, then **all** of the return values are returned via a dictionary. If some elements do not have a key, then they are numbered starting at zero.
Let's take a look at your first dictionary-based window.

```
import PySimpleGUI as sg
window = sg.Window('Simple data entry window')
layout = [
          [sg.Text('Please enter your Name, Address, Phone')],
          [sg.Text('Name', size=(15, 1)), sg.InputText('1', key='name')],
          [sg.Text('Address', size=(15, 1)), sg.InputText('2', key='address')],
          [sg.Text('Phone', size=(15, 1)), sg.InputText('3', key='phone')],
          [sg.Submit(), sg.Cancel()]
         ]

button, values = window.LayoutAndRead(layout)

sg.Popup(button, values, values['name'], values['address'], values['phone'])
```
To get the value of an input field, you use whatever value used as the `key` value as the index value. Thus to get the value of the name field, it is written as
```
values['name']
```
You will find the key field used quite heavily in most PySimpleGUI windows unless the window is very simple.

Button Return Values

The button value from a Read call will be one of 3 values:

1. The Button's text
2. The Button's key
3. None

If a button has a key set for it when it's created, then that key will be returned. If no key is set, then the button text is returned. If no button was clicked, but the window returned anyway, the button value is None.

None is returned when the user clicks the X to close a window.

If your window has an event loop where it is read over and over, remember to give your user an "out". You should always check for a None value and it's a good practice to provide an Exit button of some kind. Thus design patterns often resemble this Event Loop:

```
while True:
    button, values= window.Read()
    if button is None or button == 'Quit':
        break
```

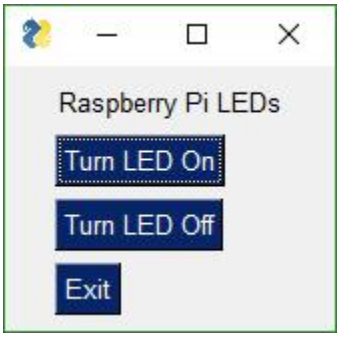
The Event Loop / Callback Functions

All GUIs have one thing in common, an "event loop" or some kind. If your program shows a single window, collects the data and then executes the primary code of the program then you likely don't need an event loop.

Event Loops are used in programs where the window ***stays open*** after button presses. The program processes button clicks and user input in a loop called the event loop. You often hear the term event loop when discussing embedded systems or on a Raspberry Pi.

Let's take a Pi demo program as an example. This program shows a GUI window, gets button presses, and uses them to control some LEDs. It loops, reading user input and doing something with it.

This little program has a typical Event Loop




```
import PySimpleGUI as sg
layout = [[sg.T('Raspberry Pi LEDs')],
          [sg.RButton('Turn LED On')],
          [sg.RButton('Turn LED Off')],
          [sg.Exit()]]
```

```
window = sg.Window('Raspberry Pi).Layout(layout)

# ---- Event Loop ---- #
while True:
    button, values = window.Read()

    # ---- Process Button Clicks ---- #
   if button is None or button == 'Exit':
        break
  if button == 'Turn LED Off':
        turn_LED_off()
   elif button == 'Turn LED On':
        turn_LED_on()

# ---- After Event Loop ---- #
sg.Popup('Done... exiting')
```

In the Event Loop we are reading the window and then doing a series of button compares to determine what to do based on the button that was clicks (value of `button` variable)
The way buttons are presented to the caller in PySimpleGUI is ***not*** how *most* GUI frameworks handle button clicks. Most GUI frameworks, including tkinter, use ***callback*** functions, a function you define would be called when a button is clicked. This requires you to write code where data is shared.

There is a more communications that have to happen between parts of your program when using callbacks. Callbacks can break your program's logic apart and scatter it. One of the larger hurdles for beginners to GUI programming are these callback functions.

PySimpleGUI was specifically designed in a way that callbacks would not be required. There is no coordination between one function and another required. You simply read your button click and take appropriate action at the same location as when you .

Whether or not this is a "proper" design for GUI programs can be debated. It's not a terrible trade-off to run your own event loop and having a functioning GUI application versus one that maybe never gets written because callback functions were too much to grasp.

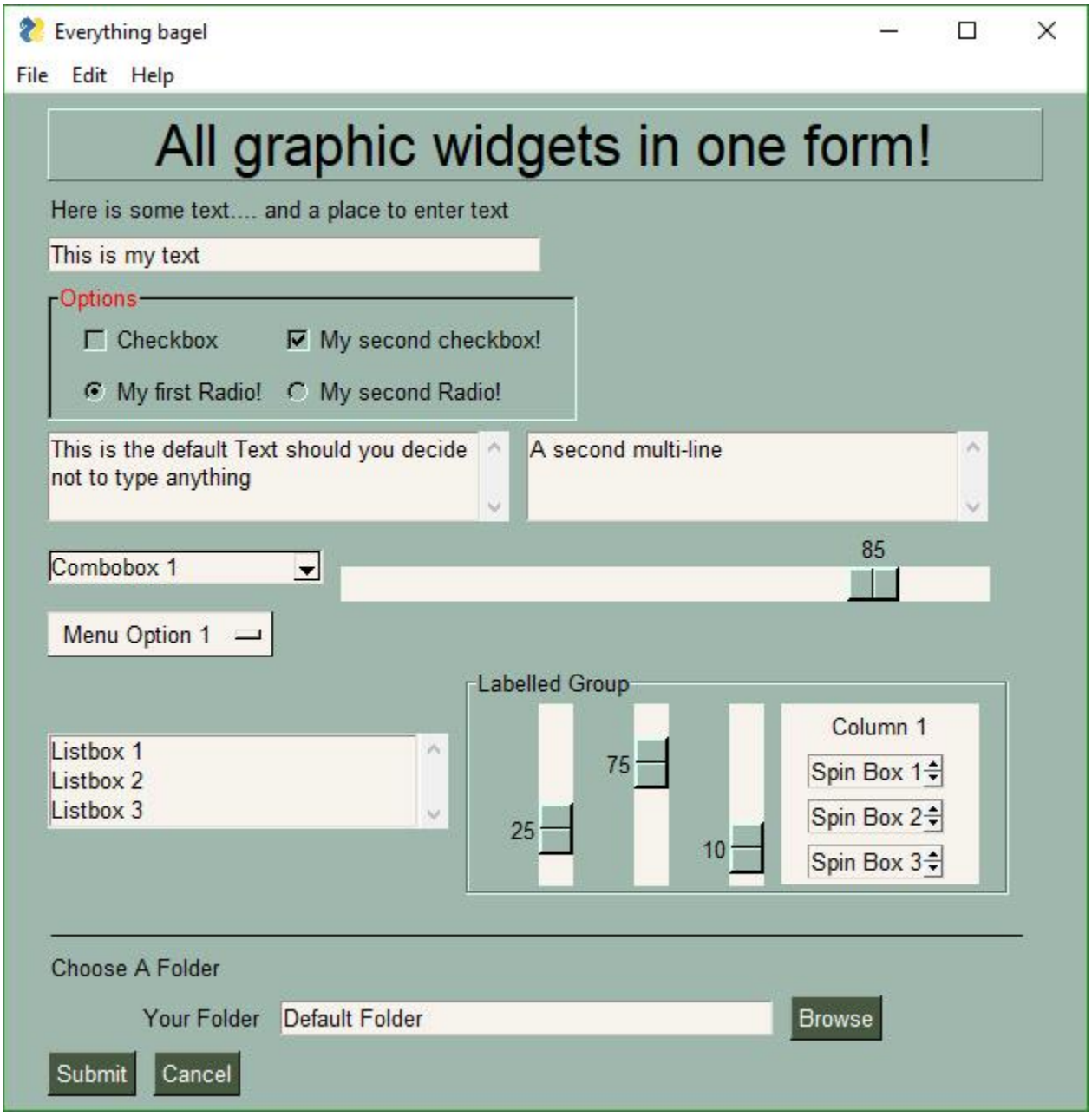
All Widgets / Elements

This code utilizes as many of the elements in one window as possible.

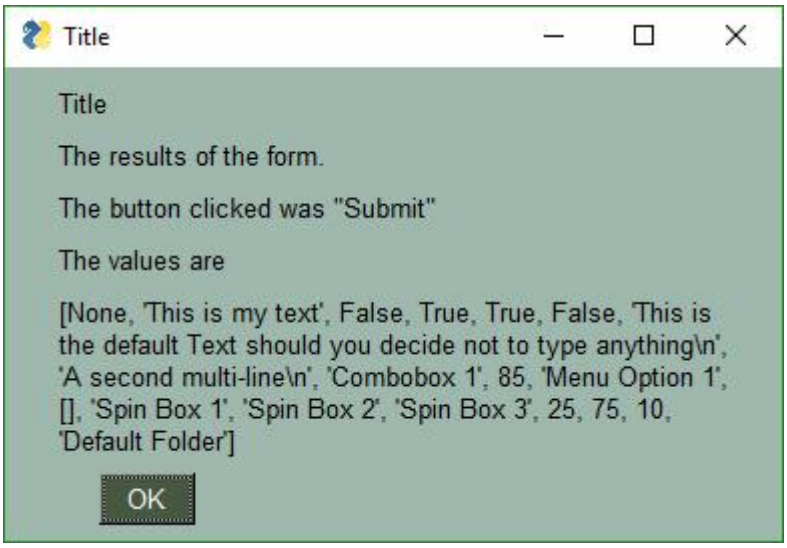
```
import PySimpleGUI as sg

sg.ChangeLookAndFeel('GreenTan')

# ------ Menu Definition ------ #
menu_def = [['File', ['Open', 'Save', 'Exit', 'Properties']],
            ['Edit', ['Paste', ['Special', 'Normal', ], 'Undo'], ],
            ['Help', 'About...'], ]

# ------ Column Definition ------ #
column1 = [[sg.Text('Column 1', background_color='#F7F3EC', justification='center',
size=(10, 1))],
```

```
          [sg.Spin(values=('Spin Box 1', '2', '3'), initial_value='Spin Box 1')],
          [sg.Spin(values=('Spin Box 1', '2', '3'), initial_value='Spin Box 2')],
          [sg.Spin(values=('Spin Box 1', '2', '3'), initial_value='Spin Box 3')]]

layout = [
    [sg.Menu(menu_def, tearoff=True)],
    [sg.Text('All graphic widgets in one window!', size=(30, 1),
justification='center', font=("Helvetica", 25), relief=sg.RELIEF_RIDGE)],
    [sg.Text('Here is some text.... and a place to enter text')],
    [sg.InputText('This is my text')],
    [sg.Frame(layout=[
    [sg.Checkbox('Checkbox', size=(10,1)),  sg.Checkbox('My second checkbox!',
default=True)],
    [sg.Radio('My first Radio!     ', "RADIO1", default=True, size=(10,1)),
sg.Radio('My second Radio!', "RADIO1")]], title='Options',title_color='red',
relief=sg.RELIEF_SUNKEN, tooltip='Use these to set flags')],
    [sg.Multiline(default_text='This is the default Text should you decide not to
type anything', size=(35, 3)),
     sg.Multiline(default_text='A second multi-line', size=(35, 3))],
    [sg.InputCombo(('Combobox 1', 'Combobox 2'), size=(20, 1)),
     sg.Slider(range=(1, 100), orientation='h', size=(34, 20), default_value=85)],
    [sg.InputOptionMenu(('Menu Option 1', 'Menu Option 2', 'Menu Option 3'))],
    [sg.Listbox(values=('Listbox 1', 'Listbox 2', 'Listbox 3'), size=(30, 3)),
     sg.Frame('Labelled Group',[[
     sg.Slider(range=(1, 100), orientation='v', size=(5, 20), default_value=25),
     sg.Slider(range=(1, 100), orientation='v', size=(5, 20), default_value=75),
     sg.Slider(range=(1, 100), orientation='v', size=(5, 20), default_value=10),
     sg.Column(column1, background_color='#F7F3EC')]])],
    [sg.Text('_'  * 80)],
    [sg.Text('Choose A Folder', size=(35, 1))],
    [sg.Text('Your Folder', size=(15, 1), auto_size_text=False,
justification='right'),
     sg.InputText('Default Folder'), sg.FolderBrowse()],
    [sg.Submit(tooltip='Click to submit this window'), sg.Cancel()]
]


window = sg.Window('Everything bagel', default_element_size=(40, 1),
grab_anywhere=False).Layout(layout)

button, values = window.Read()

sg.Popup('Title',
         'The results of the window.',
         'The button clicked was "{}"'.format(button),
         'The values are', values)
```

This is a somewhat complex window with quite a bit of custom sizing to make things line up well. This is code you only have to write once. When looking at the code, remember that what you're seeing is a list of lists. Each row contains a list of Graphical Elements that are used to create the window.



Clicking the Submit button caused the window call to return. The call to Popup resulted in this dialog box.



`Note, button value can be None`. The value for `button` will be the text that is displayed on the button element when it was created. If the user closed the window using something other than a button, then `button` will be `None`.
You can see in the Popup that the values returned are a list. Each input field in the window generates one item in the return values list. All input fields return a `string` except for Check Boxes and Radio Buttons. These return `bool`.

Building Custom windows

You will find it much easier to write code using PySimpleGUI if you use an IDE such as PyCharm. The features that show you documentation about the API call you are making will help you determine which settings you want to change, if any. In PyCharm, two commands are particularly helpful.

```
Control-Q (when cursor is on function name) brings up a box with the function
definition
Control-P (when cursor inside function call "()") shows a list of parameters and
their default values
```

Synchronous windows

The most common use of PySimpleGUI is to display and collect information from the user. The most straightforward way to do this is using a "blocking" GUI call. Execution is "blocked" while waiting for the user to close the GUI window/dialog box. You've already seen a number of examples above that use blocking windows. Anytime you see a context manager used (see the `with`statement) it's most likely a blocking window. You can examine the show calls to be sure. If the window is a non-blocking window, it must indicate that in the call to `window.show`.
NON-BLOCKING window call:

```
      window.Show(non_blocking=True)
```

Beginning a window

The first step is to create the window object using the desired window customization.

```
with Window('Everything bagel', auto_size_text=True, default_element_size=(30,1)) as
window:
```

This is the definition of the Window object:

```
def Window(title,
    default_element_size=(DEFAULT_ELEMENT_SIZE[0], DEFAULT_ELEMENT_SIZE[1]),
    default_button_element_size = (None, None),
    auto_size_text=None,
    auto_size_buttons=None,
    location=(None, None),
    font=None,
    button_color=None,Font=None,
    progress_bar_color=(None,None),
    background_color=None
    border_depth=None,
    auto_close=False,
    auto_close_duration=DEFAULT_AUTOCLOSE_TIME,
    icon=DEFAULT_WINDOW_ICON,
    force_toplevel=False
    return_keyboard_events=False,
    use_default_focus=True,
    text_justification=None,
    no_titlebar=False,
    grab_anywhere=False
    keep_on_top=False):
```

Parameter Descriptions. You will find these same parameters specified for each `Element` and some of them in `Row`specifications. The `Element` specified value will take precedence over the `Row` and `window` values.

```
   default_element_size - Size of elements in window in characters (width, height)
   default_button_element_size - Size of buttons on this window
   auto_size_text - Bool. True if elements should size themselves according to
contents. Defaults to True
   auto_size_buttons - Bool. True if button elements should size themselves according
to their text label
   location - (x,y) Location to place window in pixels
   font - Font name and size for elements of the window
   button_color - Default color for buttons (foreground, background). Can be text or
hex
   progress_bar_color - Foreground and background colors for progress bars
   background_color - Color of the window background
   border_depth - Amount of 'bezel' to put on input boxes, buttons, etc.
   auto_close - Bool.  If True window will autoclose
   auto_close_duration - Duration in seconds before window closes
   icon - .ICO file that will appear on the Task Bar and end of Title Bar
   force_top_level - Bool. If set causes a tk.Tk window to be used as primary window
rather than tk.TopLevel. Used to get around Matplotlib problem
   return_keyboard_events - if True key presses are returned as buttons
   use_default_focus - if True and no focus set, then automatically set a focus
   text_justification - Justification to use for Text Elements in this window
   no_titlebar - Create window without a titlebar
   grab_anywhere - Grab any location on the window to move the window
```

```
   keep_on_top - if True then window will always stop on top of other windows on the
screen.  Great for floating toolbars.
```

Window Location

PySimpleGUI computes the exact center of your window and centers the window on the screen. If you want to locate your window elsewhere, such as the system default of (0,0), if you have 2 ways of doing this. The first is when the window is created. Use the `location` parameter to set where the window. The second way of doing this is to use the `SetOptions` call which will set the default window location for all windows in the future.

Sizes

Note several variables that deal with "size". Element sizes are measured in characters. A Text Element with a size of 20,1 has a size of 20 characters wide by 1 character tall.

The default Element size for PySimpleGUI is `(45,1)`.
Sizes can be set at the element level, or in this case, the size variables apply to all elements in the window. Setting `size=(20,1)`in the window creation call will set all elements in the window to that size.
There are a couple of widgets where one of the size values is in pixels rather than characters. This is true for Progress Meters and Sliders. The second parameter is the 'height' in pixels.

No Titlebar

Should you wish to create cool looking windows that are clean with no windows titlebar, use the no_titlebar option when creating the window.

Be sure an provide your user an "exit" button or they will not be able to close the window! When no titlebar is enabled, there will be no icon on your taskbar for the window. Without an exit button you will need to kill via taskmanager... not fun.

Windows with no titlebar rely on the grab anywhere option to be enabled or else you will be unable to move the window.

Windows without a titlebar can be used to easily create a floating launcher.

Grab Anywhere

This is a feature unique to PySimpleGUI. The default is ENABLED.... unless the window is a non-blocking window.

It is turned off for non-blocking because there is a warning message printed out if the user closes a non-blocking window using a button with grab_anywhere enabled. There is no harm in these messages, but it may be distressing to the user. Should you wish to enable for a non-blocking window, simply get grab_anywhere = True when you create the window.

Always on top

To keep a window on top of all other windows on the screen, set keep_on_top = True when the window is created. This feature makes for floating toolbars that are very helpful and always visible on your desktop.

Window Methods (things you can do with a Window object)

There are a few methods (functions) that you will see in this document that act on Windows. The ones you will primarily be calling are:

```
window.Layout(layout) - Turns your definition of the Window into Window
window.Finalize() - creates the tkinter objects for the Window. Normally you do not call this
window.Read() - Read the Windows values and get the button / key that caused the Read to return
window.ReadNonBlocking() - Same as Read but will return right away
window.Refresh() - Use if updating elements and want to show the updates prior to the nex Read
window.Fill(values_dict) - Fill each Element with entry from the dictionary passed in
window.SaveToDisk(filename) - Save the Window's values to disk
window.LoadFromDisk(filename) - Load the Window's values from disk
window.CloseNonBlocking() - When done, for good, reading a non-blocking window
window.Disable() - Use to disable the window inpurt when opening another window on top of the primnary  Window
window.Enable() - Re-enable a Disabled window
window.FindElement(key) - Returns the element that has a matching key value
```

Elements

"Elements" are the building blocks used to create windows. Some GUI APIs use the term "Widget" to describe these graphic elements.

```
Text
Single Line Input
Buttons including these types:
    File Browse
    Folder Browse
    Calendar picker
    Date Chooser
    Read window
    Close window
    Realtime
Checkboxes
Radio Buttons
Listbox
Slider
Multi-line Text Input
Scroll-able Output
Progress Bar
Option Menu
Menu
Frame
```

```
Column
Graph
Image
Table
Tab, TabGroup
Async/Non-Blocking Windows
Tabbed windows
Persistent Windows
Redirect Python Output/Errors to scrolling Window
"Higher level" APIs (e.g. MessageBox, YesNobox, ...)
```

Common Parameters

Some parameters that you will see on almost all Elements are: key tooltip

Tooltip

Tooltips are text boxes that popup next to an element if you hold your mouse over the top of it. If you want to be extra kind to your window's user, then you can create tooltips for them by setting the parameter `tooltip` to some text string. You will need to supply your own line breaks / text wrapping. If you don't want to manually add them, then take a look at the standard library package `textwrap`. Tooltips are one of those "polish" items that really dress-up a GUI and show's a level of sophistication. Go ahead, impress people, throw some tooltips into your GUI.

Output Elements

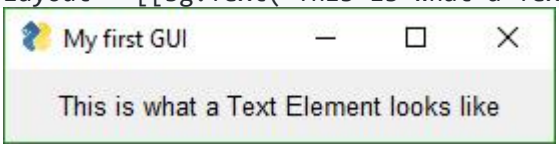
Building a window is simply making lists of Elements. Each list is a row in the overall GUI dialog box. The definition looks something like this:

```
layout = [ [row 1 element, row 1 element],
           [row 2 element, row 2 element, row 2 element] ]
```

The code is a crude representation of the GUI, laid out in text.

Text Element

```
layout = [[sg.Text('This is what a Text Element looks like')]]
```



The most basic element is the Text element. It simply displays text. Many of the 'options' that can be set for a Text element are shared by other elements.

```
Text(text
     size=(None, None)
     auto_size_text=None
     click_submits=None
     relief=None
     font=None
     text_color=None
     background_color=None
```

```
    justification=None
    pad=None
    key=None
    tooltip=None)
```

.

```
Text - The text that's displayed
size - Element's size
click_submits - if clicked will cause a read call to return they key value as the
button
relief - relief to use around the text
auto_size_text - Bool. Change width to match size of text
font - Font name and size to use
text_color - text color
background_color - background color
justification - Justification for the text. String - 'left', 'right', 'center'
pad - (x,y) amount of padding in pixels to use around element when packing
key - used to identify element.  This value will return as button if click_submits
True
tooltip - string representing tooltip
```

Some commonly used elements have 'shorthand' versions of the functions to make the code more compact. The functions `T`and `Txt` are the same as calling `Text`.
Fonts in PySimpleGUI are always in this format:

```
(font_name, point_size)
```

The default font setting is

```
("Helvetica", 10)
```

Color in PySimpleGUI are in one of two format. They can be a single color or a color pair. Buttons are an example of a color pair.

```
(foreground, background)
```

Individual colors are specified using either the color names as defined in tkinter or an RGB string of this format:

```
"#RRGGBB"
```

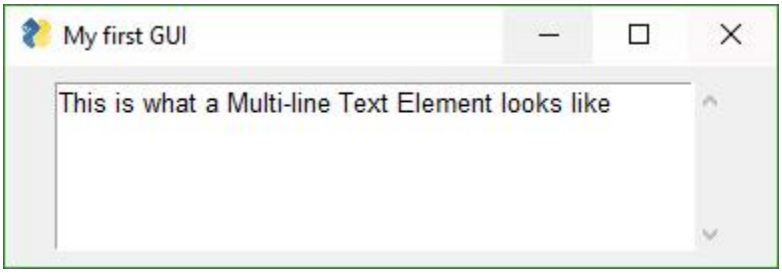
auto_size_text A `True` value for `auto_size_text`, when placed on Text Elements, indicates that the width of the Element should be shrunk do the width of the text. The default setting is True.

- List item

Shortcut functions The shorthand functions for `Text` are `Txt` and `T`

Multiline Text Element

```
layout = [[sg.Multiline('This is what a Multi-line Text Element looks like',
size=(45,5))]]
```



This Element doubles as both an input and output Element. The `DefaultText` optional parameter is used to indicate what to output to the window.

```
Multiline(default_text='',
       enter_submits = False,
          size=(None, None),
         auto_size_text=None)
```

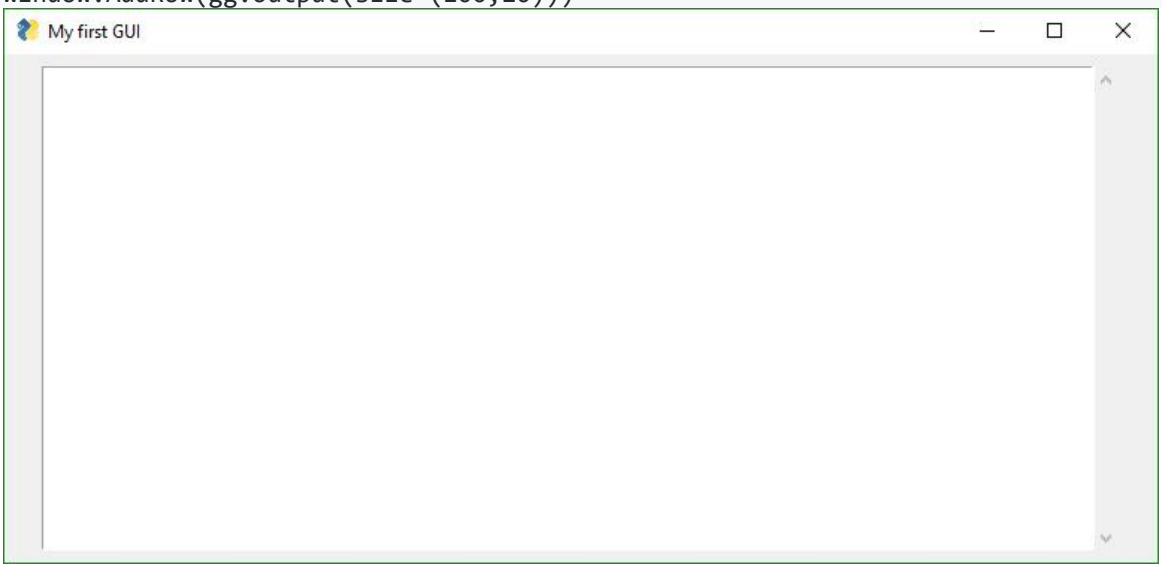
.

```
default_text - Text to display in the text box
enter_submits - Bool. If True, pressing Enter key submits window
size - Element's size
auto_size_text - Bool. Change width to match size of text
```

Output Element

Output re-routes `Stdout` to a scrolled text box. It's used with Async windows. More on this later.

```
window.AddRow(gg.Output(size=(100,20)))
```




```
Output(size=(None, None))
```

.

```
 size - Size of element (width, height) in characters
```

Input Elements

These make up the majority of the window definition. Optional variables at the Element level override the window level values (e.g. `size` is specified in the Element). All input Elements create an entry in the list of return values. A Text Input Element creates a string in the list of items returned.

Text Input Element

```
layout = [[sg.InputText('Default text')]]
```


```
  def InputText(default_text = '',
                size=(None, None),
                auto_size_text=None,
                password_char='',
                background_color=None,
                text_color=None,
                do_not_clear=False,
                key=None,
                focus=False
```

.

```
 default_text - Text initially shown in the input box
 size - (width, height) of element in characters
 auto_size_text- Bool.  True is element should be sized to fit text
 password_char - Character that will be used to replace each entered character.
Setting to a value indicates this field is a password entry field
 background_color - color to use for the input field background
 text_color - color to use for the typed text
 do_not_clear - Bool. Normally windows clear when read, turn off clearing with this
flag.
 key = Dictionary key to use for return values
 focus = Bool. True if this field should capture the focus (moves cursor to this
field)
```

There are two methods that can be called:

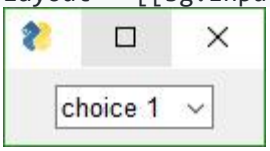
```
 InputText.Update(new_Value) - sets the input value
 Input.Text(Get() - returns the current value of the field.
```

Shorthand functions that are equivalent to `InputText` are `Input` and `In`

Combo Element

Also known as a drop-down list. Only required parameter is the list of choices. The return value is a string matching what's visible on the GUI.

```
layout = [[sg.InputCombo(['choice 1', 'choice 2'])]]
```




```
InputCombo(values,    ,
           default_value=None
```

```
            size=(None, None)
            auto_size_text=None
            background_color=None
            text_color=None
            change_submits=False
            disabled=False
            key=None
            pad=None
            tooltip=None
```

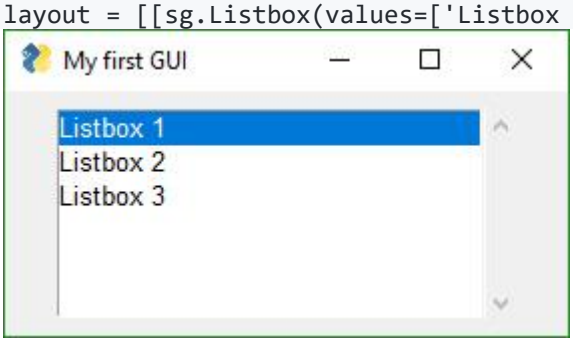
.

```
 values - Choices to be displayed. List of strings
 default_value - which value should be initially chosen
 size - (width, height) of element in characters
 auto_size_text - Bool. True if size should fit the text length
 background_color - color to use for the input field background
 text_color - color to use for the typed text
 change_submits - Bool. If set causes Read to immediately return if the selected
value changes
 disabled - Bool. If set will disable changes
 key - Dictionary key to use for return values
 pad - (x,y) Amount of padding to put around element in pixels
 tooltip -  Text string. If set, hovering over field will popup the text
```

Listbox Element

The standard listbox like you'll find in most GUIs. Note that the return values from this element will be a ***list of results, not a single result***. This is because the user can select more than 1 item from the list (if you set the right mode).

```
layout = [[sg.Listbox(values=['Listbox 1', 'Listbox 2', 'Listbox 3'], size=(30, 6))]]
```




```
    Listbox(values
            default_values=None
            select_mode=None
            change_submits=False
            bind_return_key=False
            size=(None, None)
            auto_size_text=None
            font=None
            background_color=None
            text_color=None
            key=None
            pad=None
            tooltip=None):
```

.

```
values - Choices to be displayed. List of strings
select_mode - Defines how to list is to operate.
     Choices include constants or strings:
     Constants version:
   LISTBOX_SELECT_MODE_BROWSE
   LISTBOX_SELECT_MODE_EXTENDED
   LISTBOX_SELECT_MODE_MULTIPLE
   LISTBOX_SELECT_MODE_SINGLE - the default
   Strings version:
   'browse'
   'extended'
   'multiple'
   'single'
change_submits - if True, the window read will return with a button value of ''
bind_return_key - if the focus is on the listbox and the user presses return key, or
if the user double clicks an item, then the read will return
size - (width, height) of element in characters
auto_size_text - Bool. True if size should fit the text length
background_color - color to use for the input field background
font - font to use for items in list
text_color - color to use for the typed text
key - Dictionary key to use for return values and to find element
pad - amount of padding to use when packing
tooltip - tooltip text
```

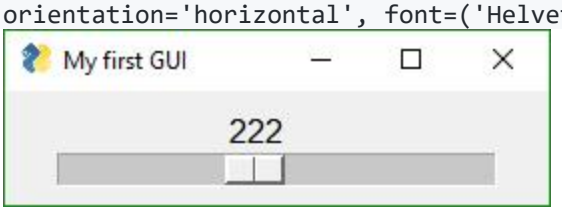
The `select_mode` option can be a string or a constant value defined as a variable. Generally speaking strings are used for these kinds of options.

ListBoxes can cause a window to return from a Read call. If the flag change_submits is set, then when a user makes a selection, the Read immediately returns. Another way ListBoxes can cause Reads to return is if the flag bind_return_key is set. If True, then if the user presses the return key while an entry is selected, then the Read returns. Also, if this flag is set, if the user double-clicks an entry it will return from the Read.

Slider Element

Sliders have a couple of slider-specific settings as well as appearance settings. Examples include the `orientation` and `range`settings.

```
layout = [[sg.Slider(range=(1,500), default_value=222, size=(20,15),
orientation='horizontal', font=('Helvetica', 12))]]
```




```
   Slider(range=(None,None),
          default_value=None,
          orientation=None,
          border_width=None,
          relief=None,
          size=(None, None),
          font=None,
          background_color = None,
          change_submits = False,
          text_color = None,
          key = None) ):
```

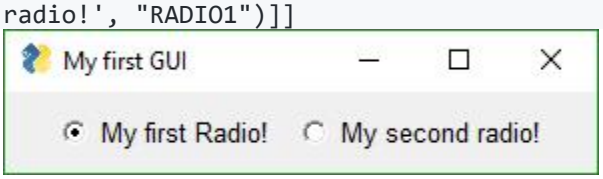
.

```
  range - (min, max) slider's range
  default_value - default setting (within range)
  orientation - 'horizontal' or 'vertical' ('h' or 'v' work)
  border_width - how deep the widget looks
  relief - relief style. Values are same as progress meter relief values.  Can be a
constant or a string:
     RELIEF_RAISED= 'raised'
     RELIEF_SUNKEN= 'sunken'
     RELIEF_FLAT= 'flat'
     RELIEF_RIDGE= 'ridge'
     RELIEF_GROOVE= 'groove'
     RELIEF_SOLID = 'solid'
   size - (width, height) of element in characters
   auto_size_text - Bool. True if size should fit the text
   background_color - color to use for the input field background
   text_color - color to use for the typed text
   change_submits - causes window read to immediately return if the checkbox value
changes
   key = Dictionary key to use for return values
```

Radio Button Element

Creates one radio button that is assigned to a group of radio buttons. Only 1 of the buttons in the group can be selected at any one time.

```
layout =  [[sg.Radio('My first Radio!', "RADIO1", default=True), sg.Radio('My second
radio!', "RADIO1")]]
```




```
Radio(text,
      group_id,
      default=False,
      size=(None, None),
      auto_size_text=None,
      font=None,
      background_color = None,
      text_color = None,
      key = None)
```

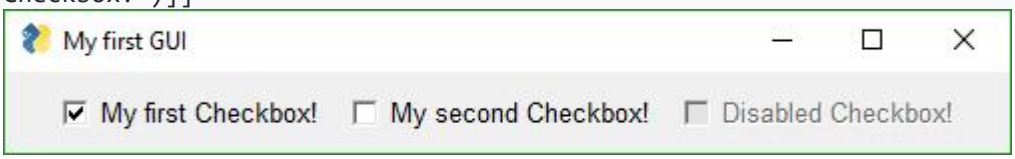
.

```
text - Text to display next to button
group_id - Groups together multiple Radio Buttons. Can be any value
default - Bool.  Initial state
size- (width, height) size of element in characters
auto_size_text - Bool.  True if should size width to fit text
font - Font type and size for text display
background_color - color to use for the background
text_color - color to use for the text
key = Dictionary key to use for return values
```

Checkbox Element

Checkbox elements are like Radio Button elements. They return a bool indicating whether or not they are checked.

```
layout =  [[sg.Checkbox('My first Checkbox!', default=True), sg.Checkbox('My second Checkbox!')]]
```




```
Checkbox(text,
         default=False,
         size=(None, None),
         auto_size_text=None,
         font=None,
         background_color = None,
         text_color = None,
         change_submits = False
         key = None):
```

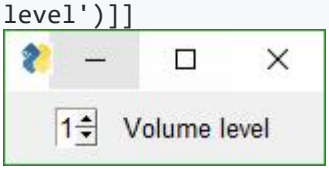
.

```
 text - Text to display next to checkbox
 default- Bool + None.  Initial state. True = Checked, False = unchecked, None = Not available (grayed out)
 size - (width, height) size of element in characters
 auto_size_text- Bool.  True if should size width to fit text
 font- Font type and size for text display
 background_color - color to use for the background
 text_color - color to use for the typed text
 change_submits - causes window read to immediately return if the checkbox value changes
 key = Dictionary key to use for return values
```

Spin Element

An up/down spinner control. The valid values are passed in as a list.

```
layout =  [[sg.Spin([i for i in range(1,11)], initial_value=1), sg.Text('Volume level')]]
```




```
Spin(values,
     intiial_value=None,
     size=(None, None),
     auto_size_text=None,
     font=None,
     background_color = None,
     text_color = None,
     key = None):
```

.

```
 values - List of valid values
 initial_value - String with initial value
 size - (width, height) size of element in characters
 auto_size_text - Bool.  True if should size width to fit text
 font - Font type and size for text display
 background_color - color to use for the background
 text_color - color to use for the typed text
 change_submits - causes window read to immediately return if the spinner value
changes
 key = Dictionary key to use for return values
```

Button Element

Buttons are the most important element of all! They cause the majority of the action to happen. After all, it's a button press that will get you out of a window, whether it be Submit or Cancel, one way or another a button is involved in all windows. The only exception is to this is when the user closes the window using the "X" in the upper corner which means no button was involved.

The Types of buttons include:

- Folder Browse
- File Browse
- Files Browse
- File SaveAs
- File Save
- Close window (normal button)
- Read window
- Realtime
- Calendar Chooser
- Color Chooser

Close window - Normal buttons like Submit, Cancel, Yes, No, etc, are "Close window" buttons. They cause the input values to be read and then the window is ***closed***, returning the values to the caller.

Folder Browse - When clicked a folder browse dialog box is opened. The results of the Folder Browse dialog box are written into one of the input fields of the window.

File Browse - Same as the Folder Browse except rather than choosing a folder, a single file is chosen.

Calendar Chooser - Opens a graphical calendar to select a date.

Color Chooser - Opens a color chooser dialog

Read window - This is a window button that will read a snapshot of all of the input fields, but does not close the window after it's clicked.

Realtime - This is another async window button. Normal button clicks occur after a button's click is released. Realtime buttons report a click the entire time the button is held down.

Most programs will use a combination of shortcut button calls (Submit, Cancel, etc), plain buttons that close the window, and ReadForm buttons that keep the window open but returns control back to the caller.

Sometimes there are multiple names for the same function. This is simply to make the job of the programmer quicker and easier.

The 3 primary windows of PySimpleGUI buttons and their names are:

1. `Button` = `SimpleButton`
2. `ReadButton` = `RButton` = `ReadFormButton` (old style... use ReadButton instead)
3. `RealtimeButton`

You will find the long-form in the older programs.

The most basic Button element call to use is `Button`

```
Button(button_text=''
       button_type=BUTTON_TYPE_CLOSES_WIN
       target=(None, None)
       tooltip=None
       file_types=(("ALL Files", "*.*"),)
       initial_folder=None
       image_filename=None
       image_size=(None, None)
       image_subsample=None
       border_width=None
       size=(None, None)
       auto_size_button=None
       button_color=None
       default_value = None
       font=None
       bind_return_key=False
       focus=False
       pad=None
       key=None):
```

Parameters

```
button_text - Text to be displayed on the button
button_type - You  should NOT be setting this directly
target - key or (row,col) target for the button
tooltip - tooltip text for the button
file_types - the filetypes that will be used to match files
initial_folder - starting path for folders and files
image_filename - image filename if there is a button image
image_size - size of button image in pixels
image_subsample - amount to reduce the size of the image
border_width - width of border around button in pixels
size - size in characters
auto_size_button - True if button size is determined by button text
button_color - (text color, backound color)
default_value - initial value for buttons that hold information
font - font to use for button text
bind_return_key - If True the return key will cause this button to fire
focus - if focus should be set to this button
pad - (x,y) padding in pixels for packing the button
```

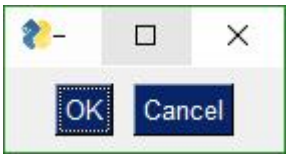
```
key - key used for finding the element
```

Pre-defined Buttons

These Pre-made buttons are some of the most important elements of all because they are used so much. They all basically do the same thing, set the button text to match the function name and set the parameters to commonly used values. If you find yourself needing to create a custom button often because it's not on this list, please post a request on GitHub. . They include:

```
OK
Ok
Submit
Cancel
Yes
No
Exit
Quit
Help
Save
SaveAs
FileBrowse
FilesBrowse
FileSaveAs
FolderBrowse
```

. layout = [[sg.OK(), sg.Cancel()]]



Button targets

The `FileBrowse`, `FolderBrowse`, `FileSaveAs` , `FilesSaveAs`, `CalendarButton`, `ColorChooserButton` buttons all fill-in values into another element located on the window. The target can be a Text Element or an InputText Element. The location of the element is specified by the `target` variable in the function call.
The Target comes in two forms.

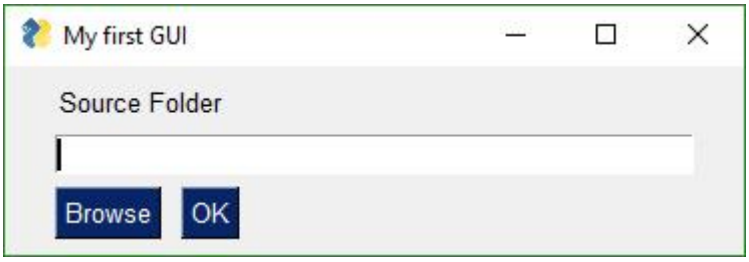
1. Key
2. (row, column)

Targets that are specified using a key will find its target element by using the target's key value. This is the "preferred" method.

If the Target is specified using (row, column) then it utilizes a grid system. The rows in your GUI are numbered starting with 0. The target can be specified as a hard coded grid item or it can be relative to the button.

The (row, col) targeting can only target elements that are in the same "container". Containers are the Window, Column and Frame Elements. A File Browse button located inside of a Column is unable to target elements outside of that Column.

The default value for `target` is `(ThisRow, -1)`. `ThisRow` is a special value that tells the GUI to use the same row as the button. The Y-value of -1 means the field one value to the left of the button. For a File or Folder Browse button, the field that it fills are generally to the left of the button is most cases. (ThisRow, -1) means the Element to the left of the button, on the same row.
If a value of `(None, None)` is chosen for the target, then the button itself will hold the information. Later the button can be queried for the value by using the button's key.
Let's examine this window as an example:



The `InputText` element is located at (1,0)... row 1, column 0. The `Browse` button is located at position (2,0). The Target for the button could be any of these values:

```
Target = (1,0)
Target = (-1,0)
```

The code for the entire window could be:

```
layout = [[sg.T('Source Folder')],
          [sg.In()],
          [sg.FolderBrowse(target=(-1, 0)), sg.OK()]]
```

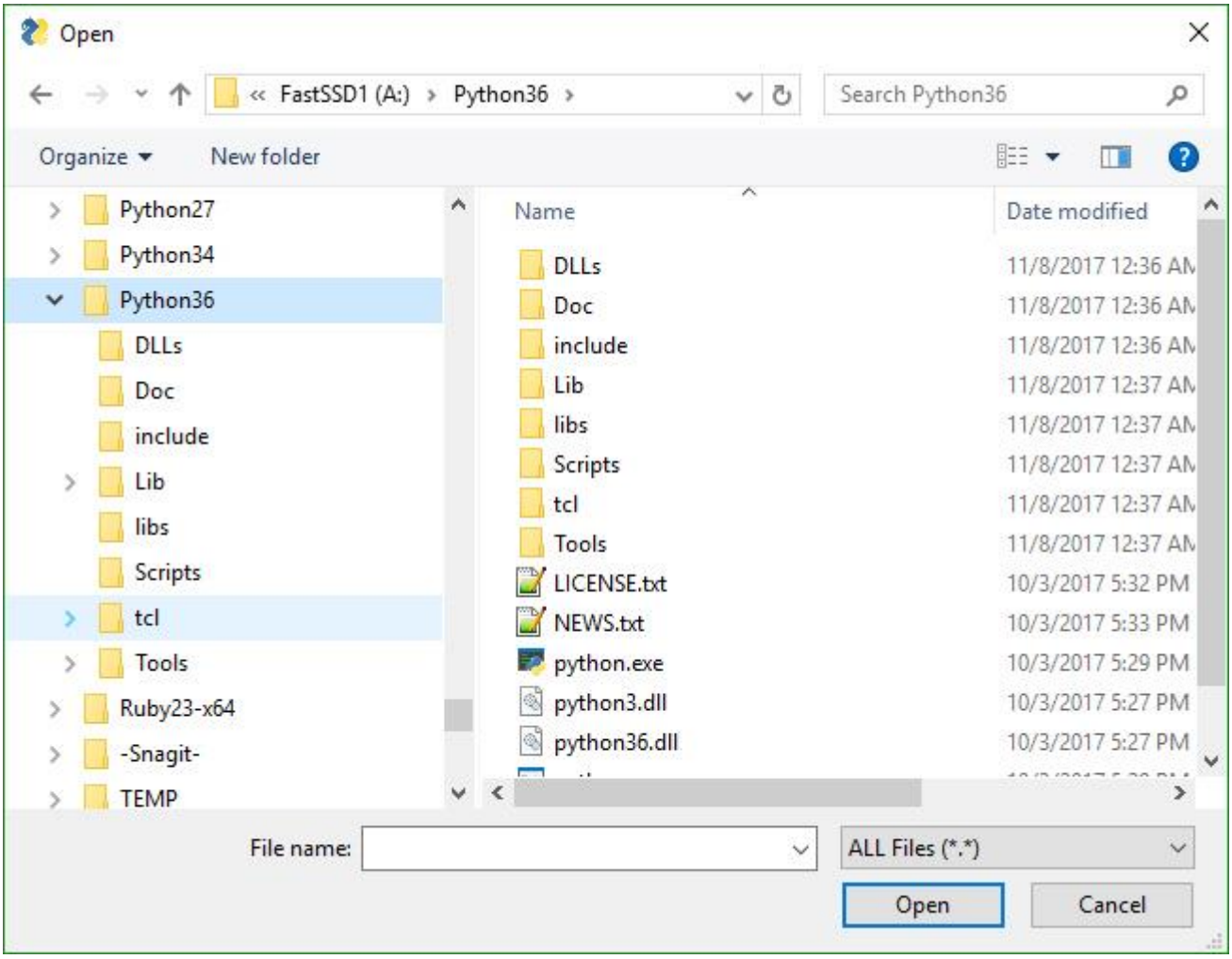
or if using keys, then the code would be:

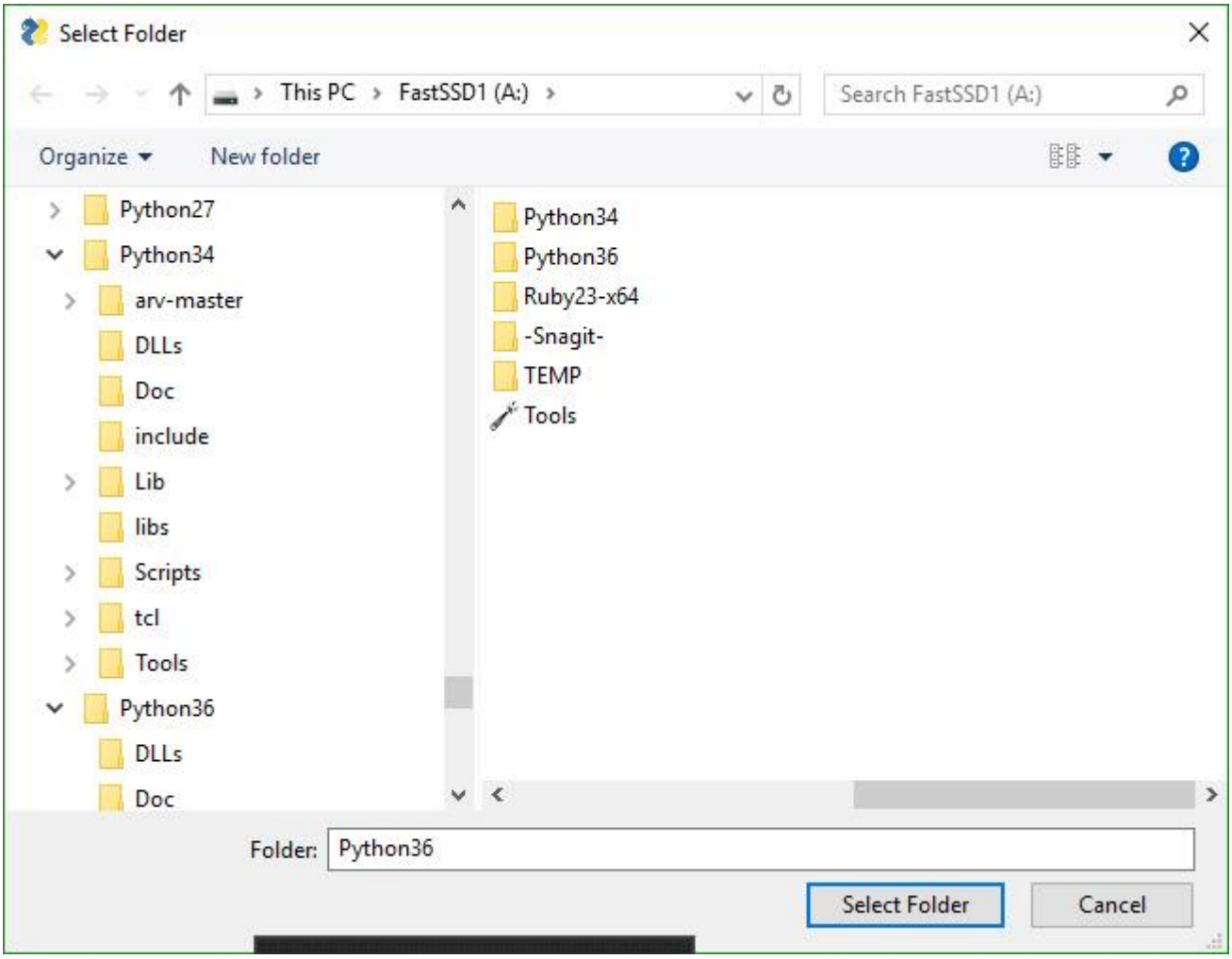
```
layout = [[sg.T('Source Folder')],
          [sg.In(key='input')],
          [sg.FolderBrowse(target='input'), sg.OK()]]
```

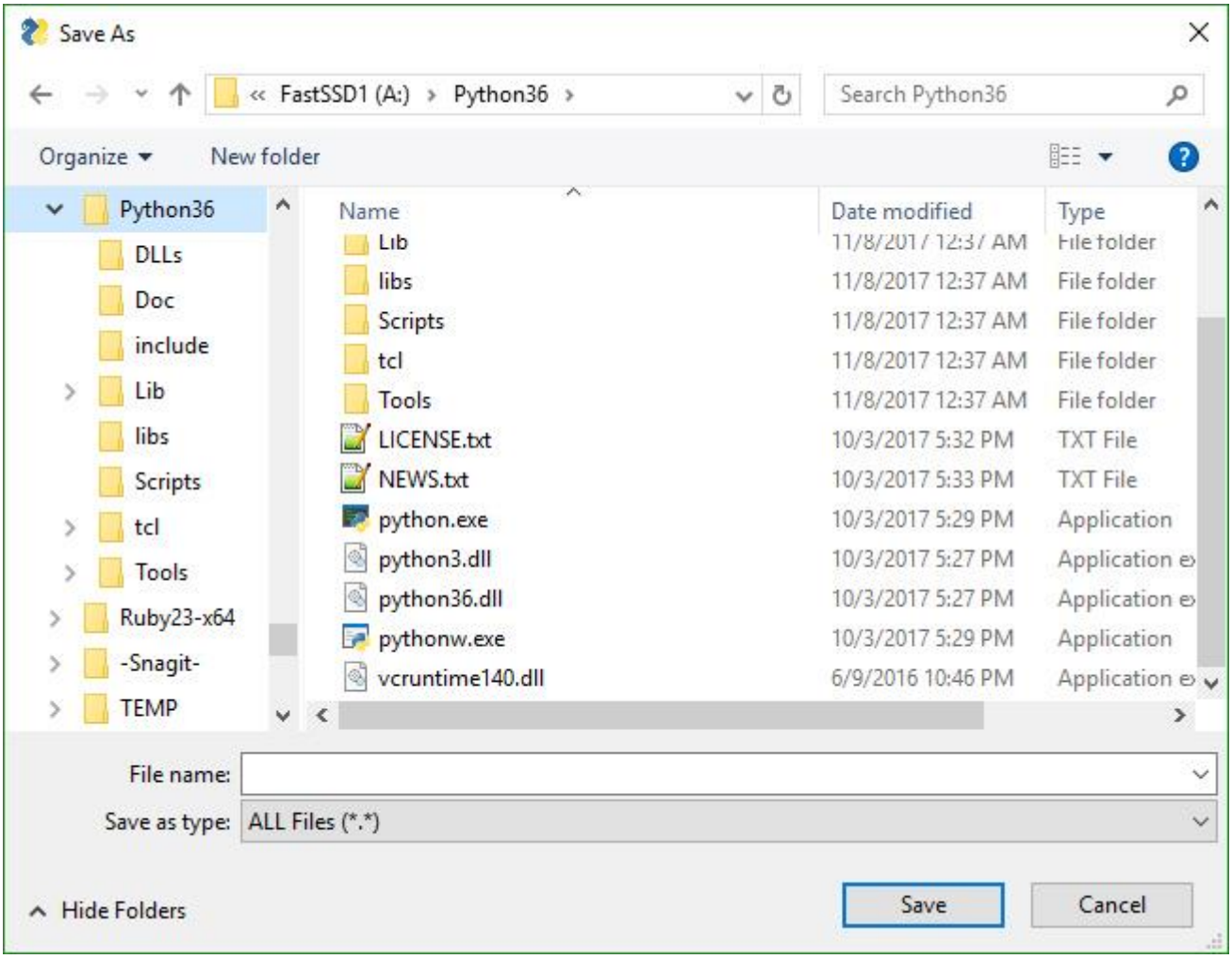
See how much easier the key method is?

Save & Open Buttons

There are 3 different types of File/Folder open dialog box available. If you are looking for a file to open, the `FileBrowse` is what you want. If you want to save a file, `SaveAs` is the button. If you want to get a folder name, then `FolderBrowse` is the button to use. To open several files at once, use the `FilesBrowse` button. It will create a list of files that are separated by ';'

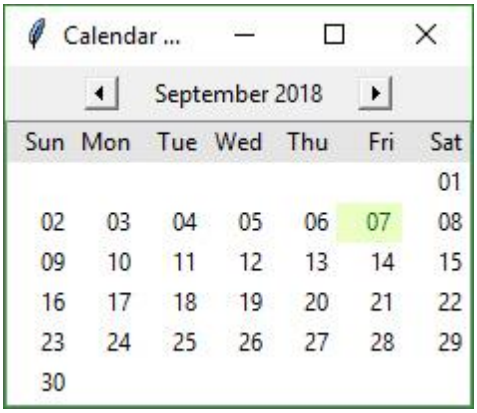

Open
« FastSSD1 (A:) › Python36 ›
Search Python36
Organize
New folder
Python27
Python34
Python36
DLLs
Doc
include
Lib
libs
Scripts
tcl
Tools
Ruby23-x64
-Snagit-
TEMP
Name
Date modified
DLLs 11/8/2017 12:36 AM
Doc 11/8/2017 12:36 AM
include 11/8/2017 12:36 AM
Lib 11/8/2017 12:37 AM
libs 11/8/2017 12:37 AM
Scripts 11/8/2017 12:37 AM
tcl 11/8/2017 12:37 AM
Tools 11/8/2017 12:37 AM
LICENSE.txt 10/3/2017 5:32 PM
NEWS.txt 10/3/2017 5:33 PM
python.exe 10/3/2017 5:29 PM
python3.dll 10/3/2017 5:27 PM
python36.dll 10/3/2017 5:27 PM
File name:
ALL Files (*.*)
Open
Cancel


Select Folder
This PC
FastSSD1 (A:)
Search FastSSD1 (A:)
Organize
New folder
Python27
Python34
arv-master
DLLs
Doc
include
Lib
libs
Scripts
tcl
Tools
Python36
DLLs
Doc
Python34
Python36
Ruby23-x64
-Snagit-
TEMP
Tools
Folder:
Python36
Select Folder
Cancel



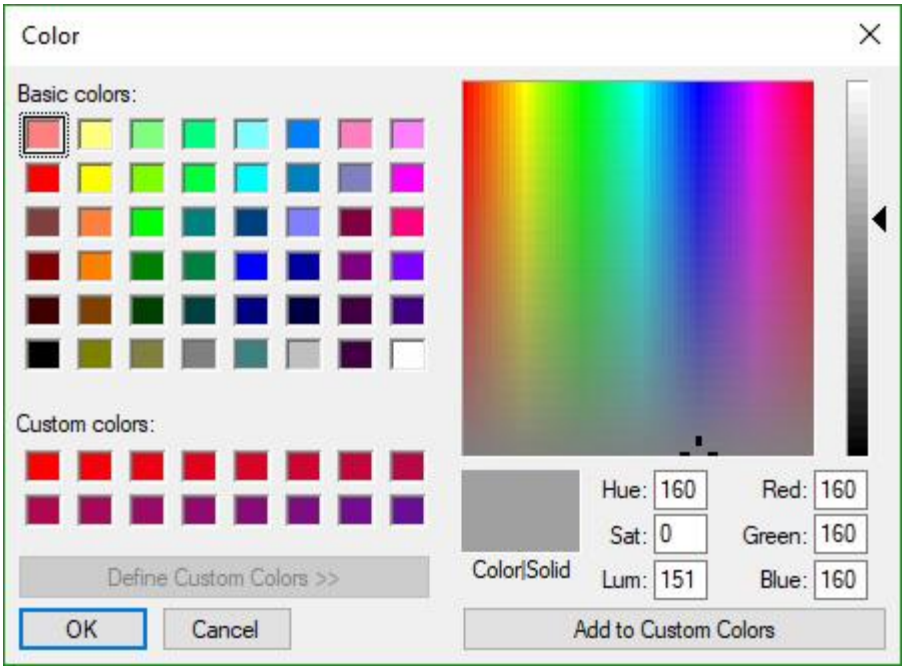
Calendar Buttons

These buttons pop up a calendar chooser window. The chosen date is returned as a string.



Color Chooser Buttons

These buttons pop up a standard color chooser window. The result is returned as a tuple. One of the returned values is an RGB hex representation.



Custom Buttons Not all buttons are created equal. A button that closes a window is different that a button that returns from the window without closing it. If you want to define your own button, you will generally do this with the Button Element `Button`, which closes the window when clicked.
layout = [[sg.Button('My Button')]]

All buttons can have their text changed by changing the `button_text` variable in the button call. It is this text that is returned when a window is read. This text will be what tells you which button is called so make it unique. Most of the convenience buttons (Submit, Cancel, Yes, etc) are all Buttons. Some that are not are `FileBrowse` , `FolderBrowse`, `FileSaveAs`. They clearly do not close the window. Instead they bring up a file or folder browser dialog box.
Button Images Now this is an exciting feature not found in many simplified packages.... images on buttons! You can make a pretty spiffy user interface with the help of a few button images.

Your button images need to be in PNG or GIF format. When you make a button with an image, set the button background to the same color as the background. There's a button color TRANSPARENT_BUTTON that you can set your button color to in order for it to blend into the background. Note that this value is currently the same as the color as the default system background on Windows.

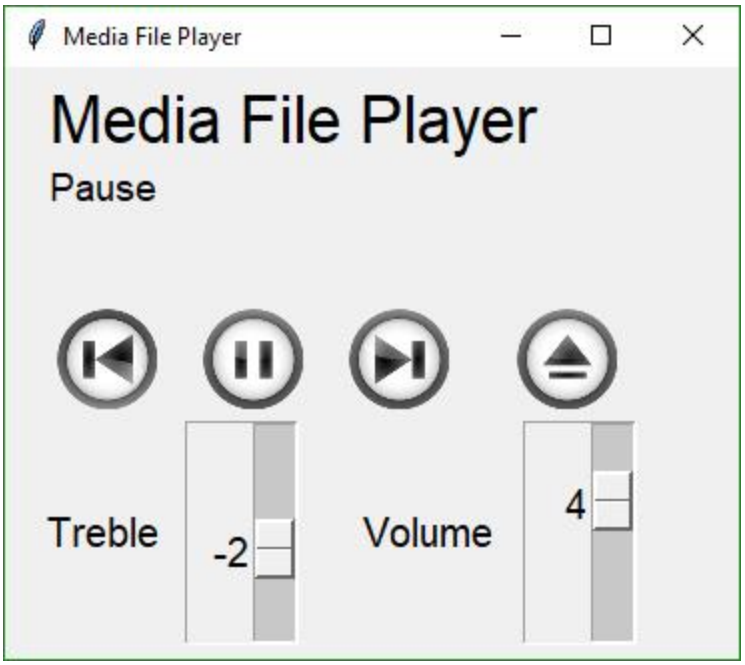
This example comes from the `Demo Media Player.py` example program. Because it's a non-blocking button, it's defined as `RButton`. You also put images on blocking buttons by using `Button`.

```
sg.RButton('Restart Song', button_color=sg.TRANSPARENT_BUTTON,
                    image_filename=image_restart, image_size=(50, 50),
image_subsample=2, border_width=0)
```

Three parameters are used for button images.

```
image_filename - Filename. Can be a relative path
image_size - Size of image file in pixels
image_subsample - Amount to divide the size by.  2 means your image will be 1/2 the
size.  3 means 1/3
```

Here's an example window made with button images.



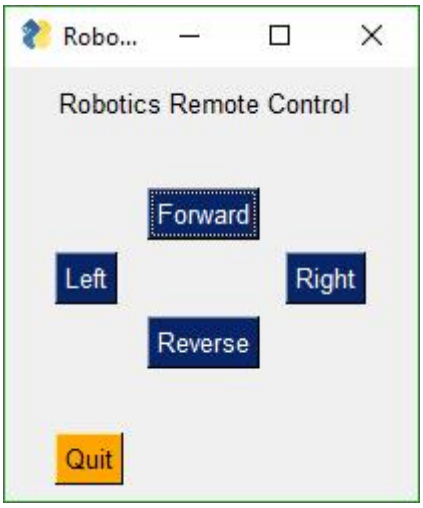
You'll find the source code in the file Demo Media Player. Here is what the button calls look like to create media player window

```
sg.RButton('Pause', button_color=sg.TRANSPARENT_BUTTON,
                  image_filename=image_pause, image_size=(50, 50), image_subsample=2,
border_width=0)
```

This is one you'll have to experiment with at this point. Not up for an exhaustive explanation.

Realtime Buttons

Normally buttons are considered "clicked" when the mouse button is let UP after a downward click on the button. What about times when you need to read the raw up/down button values. A classic example for this is a robotic remote control. Building a remote control using a GUI is easy enough. One button for each of the directions is a start. Perhaps something like this:



This window has 2 button types. There's the normal "Simple Button" (Quit) and 4 "Realtime Buttons".

Here is the code to make, show and get results from this window:

```
window = sg.Window('Robotics Remote Control', auto_size_text=True)

window_rows = [[sg.Text('Robotics Remote Control')],
             [sg.T(' '*10), sg.RealtimeButton('Forward')],
             [ sg.RealtimeButton('Left'), sg.T(' '*15), sg.RealtimeButton('Right')],
             [sg.T(' '*10), sg.RealtimeButton('Reverse')],
             [sg.T('')],
             [sg.Quit(button_color=('black', 'orange'))]
             ]

window.LayoutAndRead(window_rows, non_blocking=True)
```

Somewhere later in your code will be your main event loop. This is where you do your polling of devices, do input/output, etc. It's here that you will read your window's buttons.

```
while (True):
    # This is the code that reads and updates your window
    button, values = window.ReadNonBlocking()
    if button is not None:
        sg.Print(button)
    if button == 'Quit'  or values is None:
        break
  time.sleep(.01)
```

This loop will read button values and print them. When one of the Realtime buttons is clicked, the call to `window.ReadNonBlocking` will return a button name matching the name on the button that was depressed. It will continue to return values as long as the button remains depressed. Once released, the ReadNonBlocking will return None for buttons until a button is again clicked.

File Types The `FileBrowse` & `SaveAs` buttons have an additional setting named `file_types`. This variable is used to filter the files shown in the file dialog box. The default value for this setting is

```
FileTypes=(("ALL Files", "*.*"),)
```

This code produces a window where the Browse button only shows files of type .TXT

```
layout =  [[sg.In() ,sg.FileBrowse(file_types=(("Text Files", "*.txt"),))]]
```

The ENTER key The ENTER key is an important part of data entry for windows. There's a long tradition of the enter key being used to quickly submit windows. PySimpleGUI implements this by tying the ENTER key to the first button that closes or reads a window.

The Enter Key can be "bound" to a particular button so that when the key is pressed, it causes the window to return as if the button was clicked. This is done using the `bind_return_key` parameter in the button calls. If there are more than 1 button on a window, the FIRST button that is of type Close window or Read window is used. First is determined by scanning the window, top to bottom and left to right.

ProgressBar

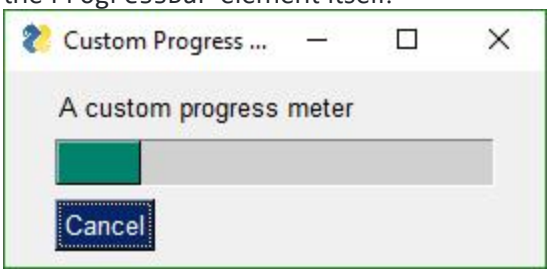
The `ProgressBar` element is used to build custom Progress Bar windows. It is HIGHLY recommended that you use OneLineProgressMeter that provides a complete progress meter solution for you. Progress Meters are not easy to work with because the windows have to be non-blocking and they are tricky to debug.
The **easiest** way to get progress meters into your code is to use the `OneLineProgressMeter` API. This consists of a pair of functions, `OneLineProgressMeter` and `OneLineProgressMeterCancel`. You can easily cancel any progress meter by calling it with the current value = max value. This will mark the meter as expired and close the window. You've already seen OneLineProgressMeter calls presented earlier in this readme.

```
sg.OneLineProgressMeter('My Meter', i+1, 1000,  'key', 'Optional message')
```

The return value for `OneLineProgressMeter` is: `True` if meter updated correctly `False` if user clicked the Cancel button, closed the window, or vale reached the max value.

Progress Mater in Your window

Another way of using a Progress Meter with PySimpleGUI is to build a custom window with a `ProgressBar` Element in the window. You will need to run your window as a non-blocking window. When you are ready to update your progress bar, you call the `UpdateBar` method for the `ProgressBar` element itself.




```
import PySimpleGUI as sg

# layout the window
layout = [[sg.Text('A custom progress meter')],
          [sg.ProgressBar(10000, orientation='h', size=(20, 20), key='progressbar')],
          [sg.Cancel()]]

# create the window`
window = sg.Window('Custom Progress Meter').Layout(layout)
```

```
progress_bar = window.FindElement('progressbar')
# loop that would normally do something useful
for i in range(10000):
    # check to see if the cancel button was clicked and exit loop if clicked
  button, values = window.ReadNonBlocking()
    if button == 'Cancel'  or values == None:
        break
  # update bar with loop value +1 so that bar eventually reaches the maximum
  progress_bar.UpdateBar(i + 1)
# done with loop... need to destroy the window as it's still open
window.CloseNonBlocking())
```

Output

The Output Element is a re-direction of Stdout. Anything "printed" will be displayed in this element.

```
Output(size=(None, None))
```

Here's a complete solution for a chat-window using an Async window with an Output Element

```
import PySimpleGUI as sg

# Blocking window that doesn't close
def ChatBot():
    layout = [[(sg.Text('This is where standard out is being routed', size=[40,
1]))],
              [sg.Output(size=(80, 20))],
              [sg.Multiline(size=(70, 5), enter_submits=True),
               sg.RButton('SEND', button_color=(sg.YELLOWS[0], sg.BLUES[0])),
               sg.Button('EXIT', button_color=(sg.YELLOWS[0], sg.GREENS[0]))]]

  window = sg.Window('Chat Window', default_element_size=(30, 2)).Layout(layout)

    # -------- Loop taking in user input and using it to query HowDoI web oracle ---
#
  while True:
        button, value = window.Read()
        if button == 'SEND':
            print(value)
        else:
            break

ChatBot()
```

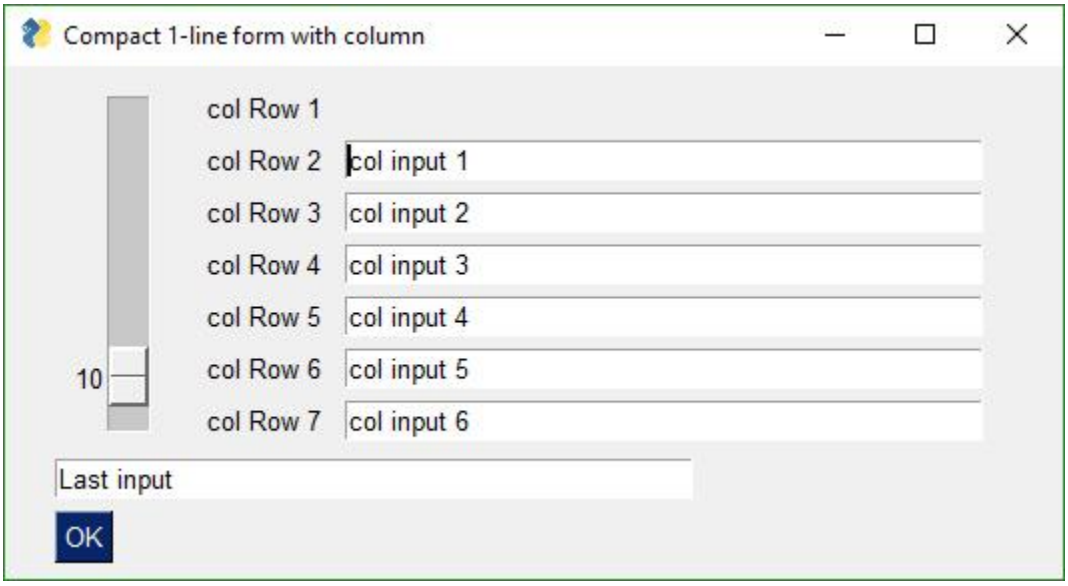
Columns

Starting in version 2.9 you'll be able to do more complex layouts by using the Column Element. Think of a Column as a window within a window. And, yes, you can have a Column within a Column if you want.

Columns are specified in exactly the same way as a window is, as a list of lists.

```
def Column(layout - the list of rows that define the layout
        background_color - color of background
        size - size of visible portion of column
        pad - element padding to use when packing
        scrollable - bool. True if should add scrollbars
```

Columns are needed when you have an element that has a height > 1 line on the left, with single-line elements on the right. Here's an example of this kind of layout:



This code produced the above window.

```
import PySimpleGUI as sg

# Demo of how columns work
# window 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 window layouts, they are a list of lists of
elements.

window = sg.Window('Columns')                                    # blank window

# Column layout
col = [[sg.Text('col Row 1')],
       [sg.Text('col Row 2'), sg.Input('col input 1')],
       [sg.Text('col Row 3'), sg.Input('col input 2')],
       [sg.Text('col Row 4'), sg.Input('col input 3')],
       [sg.Text('col Row 5'), sg.Input('col input 4')],
       [sg.Text('col Row 6'), sg.Input('col input 5')],
       [sg.Text('col Row 7'), sg.Input('col input 6')]]

layout = [[sg.Slider(range=(1,100), default_value=10, orientation='v', size=(8,20)),
sg.Column(col)],
          [sg.In('Last input')],
          [sg.OK()]]

# Display the window and get values
# If you're willing to not use the "context manager" design pattern, then it's
possible
# to collapse the window display and read down to a single line of code.
```

```
button, values = sg.Window('Compact 1-line window with column').LayoutAndRead(layout)

sg.Popup(button, values, line_width=200)
```
The Column Element has 1 required parameter and 1 optional (the layout and the background color). Setting the background color has the same effect as setting the window's background color, except it only affects the column rectangle.

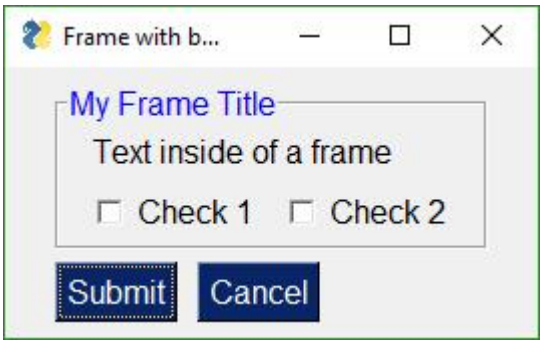
```
Column(layout, background_color=None)
```
The default background color for Columns is the same as the default window background color. If you change the look and feel of the window, the column background will match the window background automatically.

Frames (Labelled Frames, Frames with a title)

Frames work exactly the same way as Columns. You create layout that is then used to initialize the Frame.

```
def Frame(title - the label / title to put on frame
              layout - list of rows of elements the frame contains
              title_color - color of the title text
              background_color - color of background
              title_location - locations to put the title
              relief - type of relief to use
              size - size of Frame in characters. Do not use if you want frame to
autosize
              font - font to use for title
              pad - element padding  to use when packing
              border_width - how thick the line going around frame should be
              key - key used to location the element
              tooltip - tooltip text
```
This code creates a window with a Frame and 2 buttons.

```
frame_layout = [
                   [sg.T('Text inside of a frame')],
                   [sg.CB('Check 1'), sg.CB('Check 2')],
                ]
layout = [
          [sg.Frame('My Frame Title', frame_layout, font='Any 12',
title_color='blue')],
          [sg.Submit(), sg.Cancel()]
         ]

window = sg.Window('Frame with buttons', font=("Helvetica", 12)).Layout(layout)
```



Notice how the Frame layout looks identical to a window layout. A window works exactly the same way as a Column and a Frame. They all are "container elements". Elements that contain other elements.

These container Elements can be nested as deep as you want. That's a pretty spiffy feature, right? Took a lot of work so be appreciative. Recursive code isn't trivial.

Canvas Element

In my opinion, the tkinter Canvas Widget is the most powerful of the tkinter widget. While I try my best to completely isolate the user from anything that is tkinter related, the Canvas Element is the one exception. It enables integration with a number of other packages, often with spectacular results.

Matplotlib, Pyplot Integration

One such integration is with Matploplib and Pyplot. There is a Demo program written that you can use as a design pattern to get an understanding of how to use the Canvas Widget once you get it.

```
def Canvas(canvas - a tkinter canvasf if you created one. Normally not set
         background_color - canvas color
         size - size in pixels
         pad - element padding for packing
         key - key used to lookup element
         tooltip - tooltip text
```

The order of operations to obtain a tkinter Canvas Widget is:

```
figure_x, figure_y, figure_w, figure_h = fig.bbox.bounds
# define the window layout
layout = [[sg.Text('Plot test')],
          [sg.Canvas(size=(figure_w, figure_h), key='canvas')],
          [sg.OK(pad=((figure_w / 2, 0), 3), size=(4, 2))]]

# create the window and show it without the plot
window = sg.Window('Demo Application - Embedding Matplotlib In
PySimpleGUI').Layout(layout).Finalize()


# add the plot to the window
fig_photo = draw_figure(window.FindElement('canvas').TKCanvas, fig)

# show it all again and get buttons
```

```
button, values = window.Read()
```
To get a tkinter Canvas Widget from PySimpleGUI, follow these steps:

- Add Canvas Element to your window
- Layout your window
- Call `window.Finalize()` - this is a critical step you must not forget
- Find the Canvas Element by looking up using key
- Your Canvas Widget Object will be the found_element.TKCanvas
- Draw on your canvas to your heart's content
- Call `window.Read()` - Nothing will appear on your canvas until you call Read

See `Demo_Matplotlib.py` for a Recipe you can copy.

Graph Element

All you math fans will enjoy this Element... and all you non-math fans will enjoy it too.

I've found nothing to be less fun than dealing with a graphic's coordinate system from a GUI Framework. It's always upside down from what I want. (0,0) is in the upper left hand corner. In short, it's a **pain in the ass**.

Graph Element to the rescue. A Graph Element creates a pixel addressable canvas using YOUR coordinate system. *You* get to define the units on the X and Y axis.

There are 3 values you'll need to supply the Graph Element. They are:

- Size of the canvas in pixels
- The lower left (x,y) coordinate of your coordinate system
- The upper right (x,y) coordinate of your coordinate system

After you supply those values you can scribble all of over your graph by creating Graph Figures. Graph Figures are created, and a Figure ID is obtained by calling:

- DrawCircle
- DrawLine
- DrawPoint
- DrawRectangle
- DrawOval

You can move your figures around on the canvas by supplying the Figure ID the x,y amount to move.

```
graph.MoveFigure(my_circle, 10, 10)
```
This Element is relatively new and may have some parameter additions or deletions. It shouldn't break your code however.

```
def Graph( canvas_size - size of canvas in pixels
```

```
                  graph_bottom_left - the x,y location of your coordinate system's
bottom left point
                  graph_top_right - the x,y location of your coordinate system's top
right point
                  background_color - color to use for background
                  pad - element padding for pack
                  key -  key used to lookup element
                  tooltip - tooltip text
```

Table Element

Let me say up front that the Table Element has Beta status. The reason is that some of the parameters are not quite right and will change. Be warned one or two parameters may change. The `size` parameter in particular is gong to change. Currently the number of rows to allocate for the table is set by the height parameter of size. The problem is that the width is not used. The plan is to instead have a parameter named `number_of_rows` or something like it.

```
def Table(values - Your table's array
          headings - list of strings representing your headings, if you have any
          visible_column_map - list of bools. If True, column in that position is
shown.  Defaults to all columns
          col_widths - list of column widths
          def_col_width - default column width. defaults to 10
          auto_size_columns - bool. If True column widths are determined by table
contents
          max_col_width - maximum width of a column. defaults to 25
          select_mode - table rows can be selected, but doesn't currently do anything
          display_row_numbers - bool. If True shows numbers next to rows
          scrollable - if True table will be scrolled
          font - font for table entries
          justification - left, right, center
          text_color - color of text
          background_color - cell background color
          size - (None, number of rows).
          pad - element padding for packing
          key - key used to lookup element
          tooltip - tooltip text
```

Tab and Tab Group Elements

Tabs have been a part of PySimpleGUI since the initial release. However, the initial implementation applied tabs at the top level only. The entire window had to be tabbed. There with other limitations that came along with that implementation. That all changed in version 3.8.0 with the new elements - Tab and TabGroup. The old implementation of Tabs was removed in version 3.8.0 as well.

Tabs are another "Container Element". The other Container Elements include:

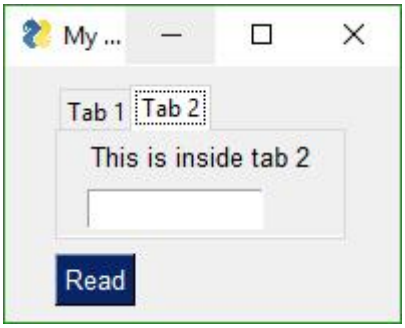
- Frame
- Column

You layout a Frame in exactly the same way as a Frame or Column elements, by passing in a list of elements.

How you place a Tab into a Window is different than Graph or Frame elements. You cannot place a tab directly into a Window's layout. It much first be placed into a TabGroup. The TabGroup can then be placed into the Window.

Let's look at this Window as an example:

View of second tab:



First we have the Tab layout definitions. They mirror what you see in the screen shots. Tab 1 has 1 Text Element in it. Tab 2 has a Text and an Input Element.

```
tab1_layout =  [[sg.T('This is inside tab 1')]]

tab2_layout = [[sg.T('This is inside tab 2')],
               [sg.In(key='in')]]
```

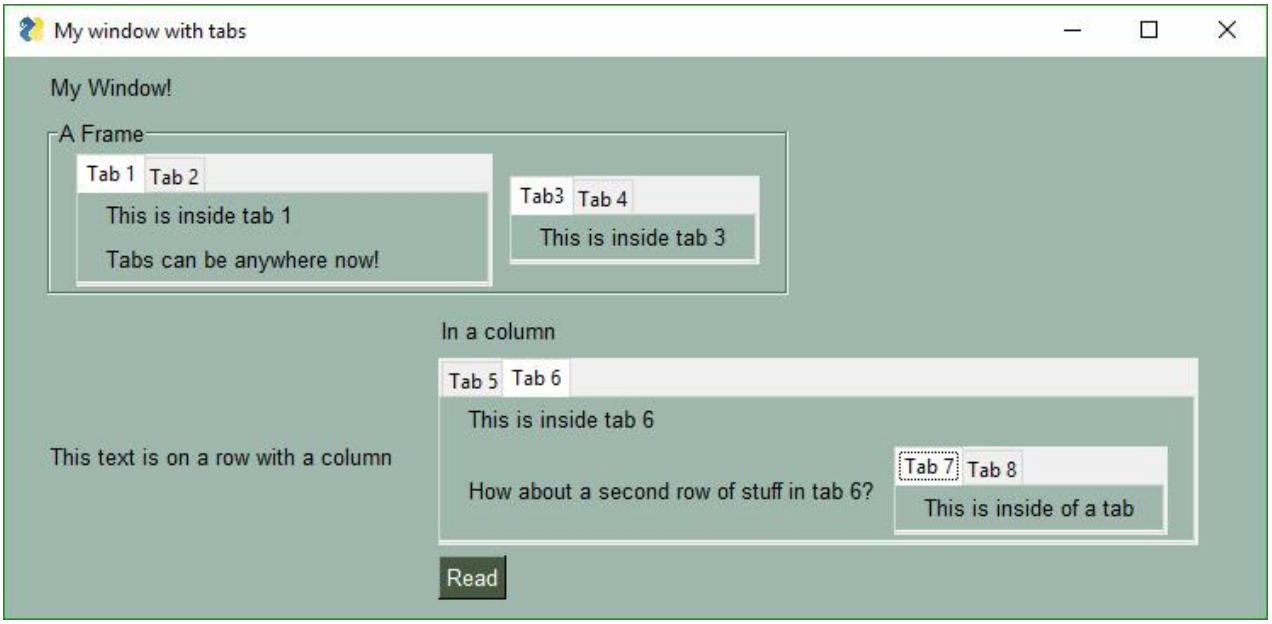
The layout for the entire window looks like this:

```
layout = [[sg.TabGroup([[sg.Tab('Tab 1', tab1_layout), sg.Tab('Tab 2',
tab2_layout)]])],
          [sg.RButton('Read')]]
```

The Window layout has the TabGroup and within the tab Group are the two Tab elements.

One important thing to notice about all of these container Elements... they all take a "list of lists" at the layout. They all have a layout that starts with `[[`
You will want to keep this `[[ ]]` construct in your head a you're debugging your tabbed windows. It's easy to overlook one or two necessary ['s
As mentioned earlier, the old-style Tabs were limited to being at the Window-level only. In other words, the tabs were equal in size to the entire window. This is not the case with the "new-style" tabs. This is why you're not going to be upset when you discover your old code no longer works with the new PySimpleGUI release. It'll be worth the few moments it'll take to convert your code.

Check out what's possible with the NEW Tabs!



Check out Tabs 7 and 8. We've got a Window with a Column containing Tabs 5 and 6. On Tab 6 are... Tabs 7 and 8.

As of Release 3.8.0, not all of *options* shown in the API definitions of the Tab and TabGroup Elements are working. They are there as placeholders.

The definition of a TabGroup is

```
 TabGroup(layout,
                  title_color=None
                  background_color=None
                  font=None
                  pad=None
                  border_width=None
                  change_submits = False
                  key=None
                  tooltip=None)
```

The definition of a Tab Element is

```
Tab(title,
    layout,
    title_color=None,
    background_color=None,
    font=None,
    pad=None
    border_width=None
    key=None
    tooltip=None)
```

Reading Tab Groups

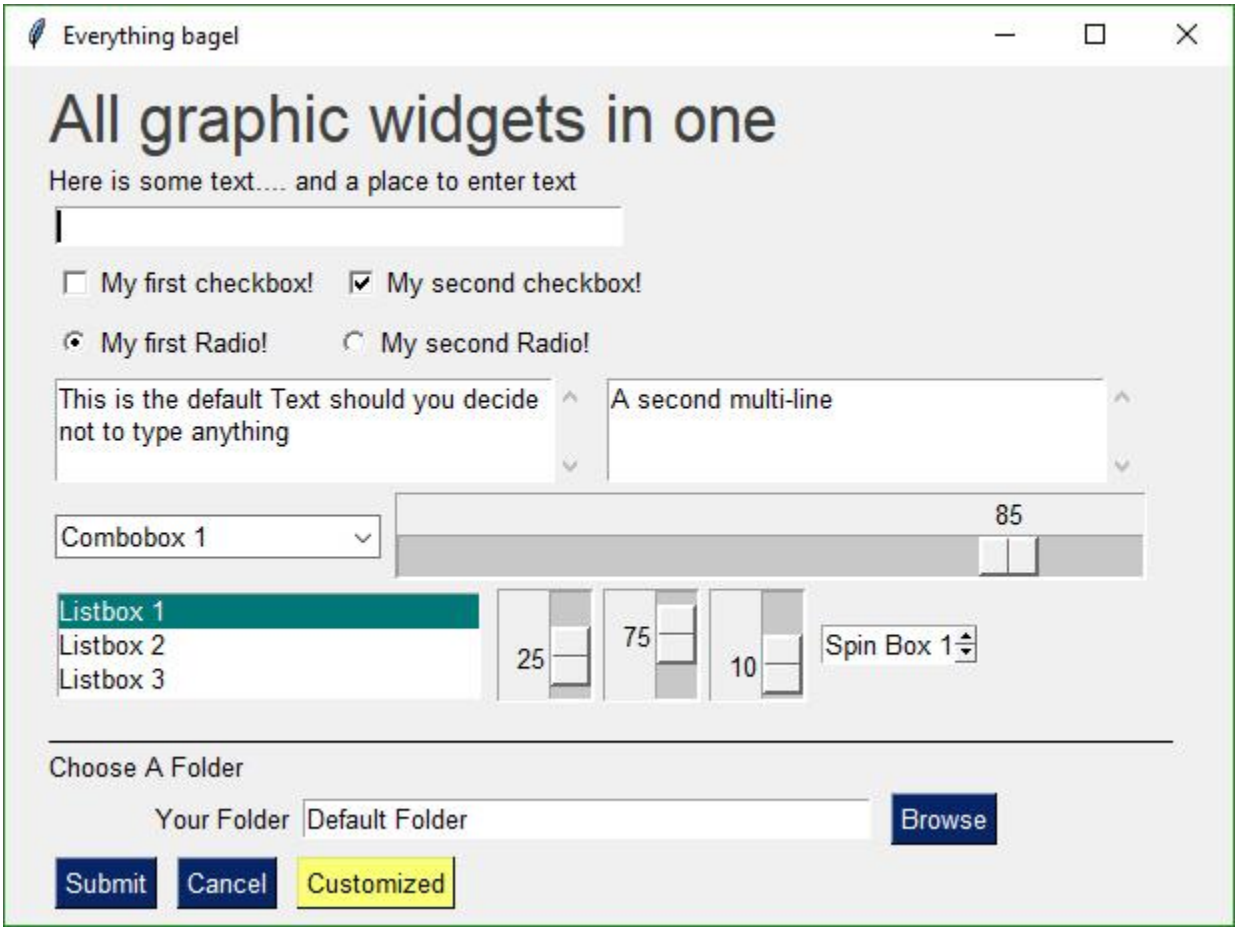
Tab Groups now return a value when a Read returns. They return which tab is currently selected. There is also a change_submits parameter that can be set that causes a Read to return if a Tab in that

group is selected / changed. The key or title belonging to the Tab that was switched to will be returned as the value

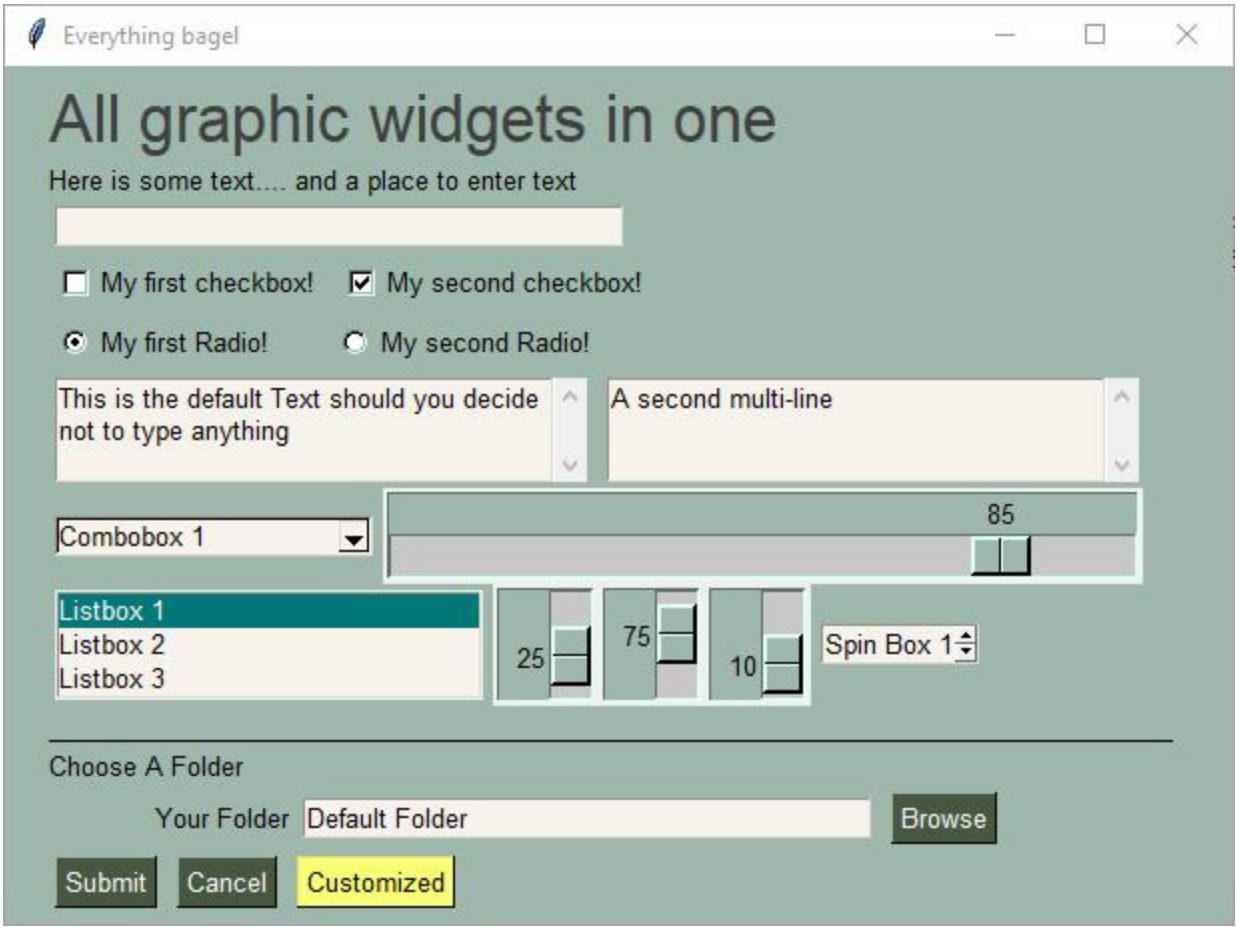
Colors

Starting in version 2.5 you can change the background colors for the window and the Elements.

Your windows can go from this:



to this... with one function call...



While you can do it on an element by element or window level basis, the easiest way, by far, is a call to `SetOptions`.
Be aware that once you change these options they are changed for the rest of your program's execution. All of your windows will have that look and feel, until you change it to something else (which could be the system default colors.

This call sets all of the different color options.

```
SetOptions(background_color='#9FB8AD',
           text_element_background_color='#9FB8AD',
           element_background_color='#9FB8AD',
           scrollbar_color=None,
           input_elements_background_color='#F7F3EC',
           progress_meter_color = ('green', 'blue')
           button_color=('white','#475841'))
```

Global Settings

Global Settings Let's have some fun customizing! Make PySimpleGUI look the way you want it to look. You can set the global settings using the function `PySimpleGUI.SetOptions`. Each option has an optional parameter that's used to set it.

```
SetOptions(icon=None
        button_color=(None,None)
        element_size=(None,None),
```

```
        margins=(None,None),
        element_padding=(None,None)
        auto_size_text=None
        auto_size_buttons=None
        font=None
        border_width=None
        slider_border_width=None
        slider_relief=None
        slider_orientation=None
        autoclose_time=None
        message_box_line_width=None
        progress_meter_border_depth=None
        progress_meter_style=None
        progress_meter_relief=None
        progress_meter_color=None
        progress_meter_size=None
        text_justification=None
        text_color=None
        background_color=None
        element_background_color=None
        text_element_background_color=None
        input_elements_background_color=None
        element_text_color=None
        input_text_color=None
        scrollbar_color=None, text_color=None
        debug_win_size=(None,None)
        window_location=(None,None)
        tooltip_time = None
```

Explanation of parameters

```
         icon - filename of icon used for taskbar and title bar
         button_color - button color (foreground, background)
         element_size - element size (width, height) in characters
         margins - tkinter margins around outsize
         element_padding - tkinter padding around each element
         auto_size_text - autosize the elements to fit their text
         auto_size_buttons - autosize the buttons to fit their text
         font - font used for elements
         border_width - amount of bezel or border around sunken or raised elements
         slider_border_width - changes the way sliders look
         slider_relief - changes the way sliders look
         slider_orientation - changes orientation of slider
         autoclose_time - time in seconds for autoclose boxes
         message_box_line_width - number of characers in a line of text in message
boxes
         progress_meter_border_depth - amount of border around raised or lowered
progress meters
         progress_meter_style - style of progress meter as defined by tkinter
         progress_meter_relief - relief style
         progress_meter_color - color of the bar and background of progress meters
         progress_meter_size - size in (characters, pixels)
         background_color - Color of the main window's background
         element_background_color - Background color of the elements
         text_element_background_color - Text element background color
         input_elements_background_color - Input fields background color
         element_text_color - Text color of elements that have text, like Radio
Buttons
         input_text_color - Color of the text that you type in
```

```
        scrollbar_color - Color for scrollbars (may not always work)
        text_color - Text element default text color
        text_justification - justification to use on Text Elements. Values are
strings - 'left', 'right', 'center'
        debug_win_size - size of the Print output window
        window_location - location on the screen (x,y) of window's top left cornder
        tooltip_time - time in milliseconds to wait before showing a tooltip.
Default is 400ms
```

These settings apply to all windows `SetOptions`. The Row options and Element options will take precedence over these settings. Settings can be thought of as levels of settings with the window-level being the highest and the Element-level the lowest. Thus the levels are:

- window level
- Row level
- Element level

Each lower level overrides the settings of the higher level. Once settings have been changed, they remain changed for the duration of the program (unless changed again).

Persistent windows (Window stays open after button click)

There are 2 ways to keep a window open after the user has clicked a button. One way is to use non-blocking windows (see the next section). The other way is to use buttons that 'read' the window instead of 'close' the window when clicked. The typical buttons you find in windows, including the shortcut buttons, close the window. These include OK, Cancel, Submit, etc. The Button Element also closes the window.

The `RButton` Element creates a button that when clicked will return control to the user, but will leave the window open and visible. This button is also used in Non-Blocking windows. The difference is in which call is made to read the window. The `Read`call will block, the `ReadNonBlocking` will not block.

Asynchronous (Non-Blocking) windows

So you want to be a wizard do ya? Well go boldly!

Use async windows sparingly. It's possible to have a window that appears to be async, but it is not. **Please** try to find other methods before going to async windows. The reason for this plea is that async windows poll tkinter over and over. If you do not have a sleep in your loop, you will eat up 100% of the CPU time.

When to use a non-blocking window:

- A media file player like an MP3 player
- A status dashboard that's periodically updated
- Progress Meters - when you want to make your own progress meters
- Output using print to a scrolled text element. Good for debugging.

If your application doesn't follow the basic design pattern at one of those, then it shouldn't be executed as a non-blocking window.

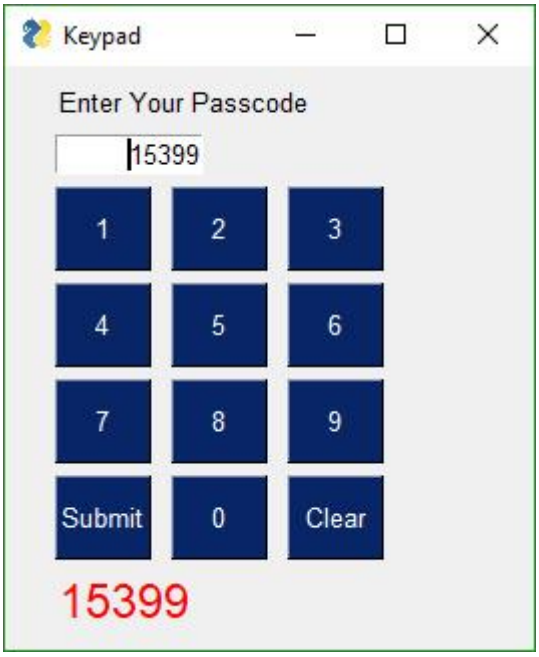
Instead of ReadNonBlocking --- Use `change_submits = True` or return_keyboard_events = True

Any time you are thinking "I want an X Element to cause a Y Element to do something", then you want to use the `change_submits` option.

Instead of polling, try options that cause the window to return to you. By using non-blocking windows, you are *polling*. You can indeed create your application by polling. It will work. But you're going to be maxing out your processor and may even take longer to react to an event than if you used another technique.

Examples

One example is you have an input field that changes as you press buttons on an on-screen keypad.



Periodically Calling`ReadNonBlocking`

Periodically "refreshing" the visible GUI. The longer you wait between updates to your GUI the more sluggish your windows will feel. It is up to you to make these calls or your GUI will freeze.

There are 2 methods of interacting with non-blocking windows.

1. Read the window just as you would a normal window
2. "Refresh" the window's values without reading the window. It's a quick operation meant to show the user the latest values

With asynchronous windows the window is shown, user input is read, but your code keeps right on chugging. YOUR responsibility is to call `PySimpleGUI.ReadNonBlocking` on a periodic basis. Once a second or more will produce a reasonably snappy GUI.

Exiting a Non-Blocking window

It's important to always provide a "way out" for your user. Make sure you have provided a button or some other mechanism to exit. Also be sure to check for closed windows in your code. It is possible for a window to look closed, but continue running your event loop.

Typically when reading a window you check `if Button is None` to determine if a window was closed. With NonBlocking windows, buttons will be None unless a button or a key was returned. The way you determine if a window was closed in a non-blocking window is to check **both** the button and the values are None. Since button is normally None, you only need to test for `value is None` in your code.
The proper code to check if the user has exited the window will be a polling-loop that looks something like this:

```
while True:
   button, values = window.ReadNonBlocking()
   if values is None or button == 'Quit':
      break
```

We're going to build an app that does the latter. It's going to update our window with a running clock.

The basic flow and functions you will be calling are: Setup

```
   window = Window()
   window_rows = .....
   window.LayoutAndRead(window_rows, non_blocking=True)
```

Periodic refresh

```
window.ReadNonBlocking()   or       window.Refresh()
```

If you need to close the window

```
window.CloseNonBlocking()
```

Rather than the usual `window.LayoutAndRead()` call, we're manually adding the rows (doing the layout) and then showing the window. After the window is shown, you simply call `window.ReadNonBlocking()` every now and then.
When you are ready to close the window (assuming the window wasn't closed by the user or a button click) you simply call `window.CloseNonBlocking()`
Example - Running timer that updates See the sample code on the GitHub named Demo Media Player for another example of Async windows. We're going to make a window and update one of the elements of that window every .01 seconds. Here's the entire code to do that.

```
import PySimpleGUI as sg
import time

# window that doesn't block
# Make a window, but don't use context manager
window = sg.Window('Running Timer', auto_size_text=True)

# Create the layout
window_rows = [[sg.Text('Non-blocking GUI with updates')],
               [sg.Text('', size=(8, 2), font=('Helvetica', 20), key='output')    ],
               [sg.Button('Quit')]]
```

```
# Layout the rows of the window and perform a read. Indicate the window is non-
blocking!
window.LayoutAndRead(window_rows, non_blocking=True)

#
# Some place later in your code...
# You need to perform a ReadNonBlocking on your window every now and then or
# else it won't refresh
#

for i in range(1, 1000):
    window.FindElement('output').Update('{:02d}:{:02d}.{:02d}'.format(*divmod(int(i /
100), 60), i % 100))
    button, values = window.ReadNonBlocking()
    if values is None or button == 'Quit':
        break
    time.sleep(.01)
else:
    window.CloseNonBlocking()
```

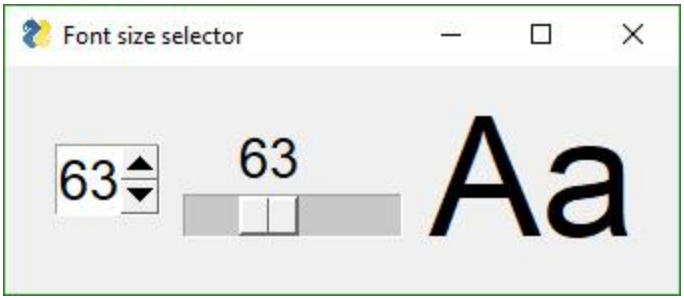
What we have here is the same sequence of function calls as in the description. Get a window, add rows to it, show the window, and then refresh it every now and then.

The new thing in this example is the call use of the Update method for the Text Element. The first thing we do inside the loop is "update" the text element that we made earlier. This changes the value of the text field on the window. The new value will be displayed when `window.ReadNonBlocking()` is called. if you want to have the window reflect your changes immediately, call `window.Refresh()`. Note the `else` statement on the for loop. This is needed because we're about to exit the loop while the window is still open. The user has not closed the window using the X nor a button so it's up to the caller to close the window using `CloseNonBlocking`.

Updating Elements (changing elements in active window)

Persistent windows remain open and thus continue to interact with the user after the Read has returned. Often the program wishes to communicate results (output information) or change an Element's values (such as populating a List Element).

The way this is done is via an Update method that is available for nearly all of the Elements. Here is an example of a program that uses a persistent window that is updated.



In some programs these updates happen in response to another Element. This program takes a Spinner and a Slider's input values and uses them to resize a Text Element. The Spinner and Slider are on the left, the Text element being changed is on the right.

```
# Testing async window, see if can have a slider
# that adjusts the size of text displayed

import PySimpleGUI as sg
fontSize = 12
layout = [[sg.Spin([sz for sz in range(6, 172)], font=('Helvetica 20'),
initial_value=fontSize, change_submits=True, key='spin'),
           sg.Slider(range=(6,172), orientation='h', size=(10,20),
           change_submits=True, key='slider', font=('Helvetica 20')),
           sg.Text("Aa", size=(2, 1), font="Helvetica "  + str(fontSize),
key='text')]]

sz = fontSize
window = sg.Window("Font size selector", grab_anywhere=False).Layout(layout)
# Event Loop
while True:
    button, values= window.Read()
    if button is None:
        break
    sz_spin = int(values['spin'])
    sz_slider = int(values['slider'])
    sz = sz_spin if sz_spin != fontSize else sz_slider
    if sz != fontSize:
        fontSize = sz
        font = "Helvetica "  + str(fontSize)
        window.FindElement('text').Update(font=font)
        window.FindElement('slider').Update(sz)
        window.FindElement('spin').Update(sz)

print("Done.")
```

Inside the event loop we read the value of the Spinner and the Slider using those Elements' keys. For example, `values['slider']` is the value of the Slider Element.
This program changes all 3 elements if either the Slider or the Spinner changes. This is done with these statements:

```
    window.FindElement('text').Update(font=font)
    window.FindElement('slider').Update(sz)
    window.FindElement('spin').Update(sz)
```

Remember this design pattern because you will use it OFTEN if you use persistent windows.

It works as follows. The call to `window.FindElement` returns the Element object represented by they provided `key`. This element is then updated by calling it's `Update` method. This is another example of Python's "chaining" feature. We could write this code using the long-form:

```
text_element = window.FindElement('text')
text_element.Update(font=font)
```

The takeaway from this exercise is that keys are key in PySimpleGUI's design. They are used to both read the values of the window and also to identify elements. As already mentioned, they are used as targets in Button calls.

Keyboard & Mouse Capture

Beginning in version 2.10 you can capture keyboard key presses and mouse scroll-wheel events. Keyboard keys can be used, for example, to detect the page-up and page-down keys for a PDF

viewer. To use this feature, there's a boolean setting in the Window
call `return_keyboard_events` that is set to True in order to get keys returned along with buttons. Keys and scroll-wheel events are returned in exactly the same way as buttons.

For scroll-wheel events, if the mouse is scrolled up, then the `button` text will be `MouseWheel:Up`. For downward scrolling, the text returned is `MouseWheel:Down`
Keyboard keys return 2 types of key events. For "normal" keys (a,b,c, etc), a single character is returned that represents that key. Modifier and special keys are returned as a string with 2 parts:

```
Key Sym:Key Code
```

Key Sym is a string such as 'Control_L'. The Key Code is a numeric representation of that key. The left control key, when pressed will return the value 'Control_L:17'

```
import PySimpleGUI as sg

# Recipe for getting keys, one at a time as they are released
# If want to use the space bar, then be sure and disable the "default focus"

with sg.Window("Keyboard Test", return_keyboard_events=True, use_default_focus=False)
as window:
    text_elem = sg.Text("", size=(18,1))
    layout = [[sg.Text("Press a key or scroll mouse")],
              [text_elem],
              [sg.Button("OK")]]

    window.Layout(layout)
    # -------- Loop taking in user input --- #
  while True:
        button, value = window.ReadNonBlocking()

        if button == "OK"  or (button is None and value is None):
            print(button, "exiting")
            break
        if button is not None:
            text_elem.Update(button)
```

You want to turn off the default focus so that there no buttons that will be selected should you press the spacebar.

Realtime Keyboard Capture

Use realtime keyboard capture by calling

```
import PySimpleGUI as sg

with sg.Window("Realtime Keyboard Test", return_keyboard_events=True,
use_default_focus=False) as window:
    layout = [[sg.Text("Hold down a key")],
              [sg.Button("OK")]]

    window.Layout(layout)

    while True:
        button, value = window.ReadNonBlocking()
```

```
        if button == "OK":
            print(button, value, "exiting")
            break
      if button is not None:
            print(button)
        elif value is None:
            break
```

Menus

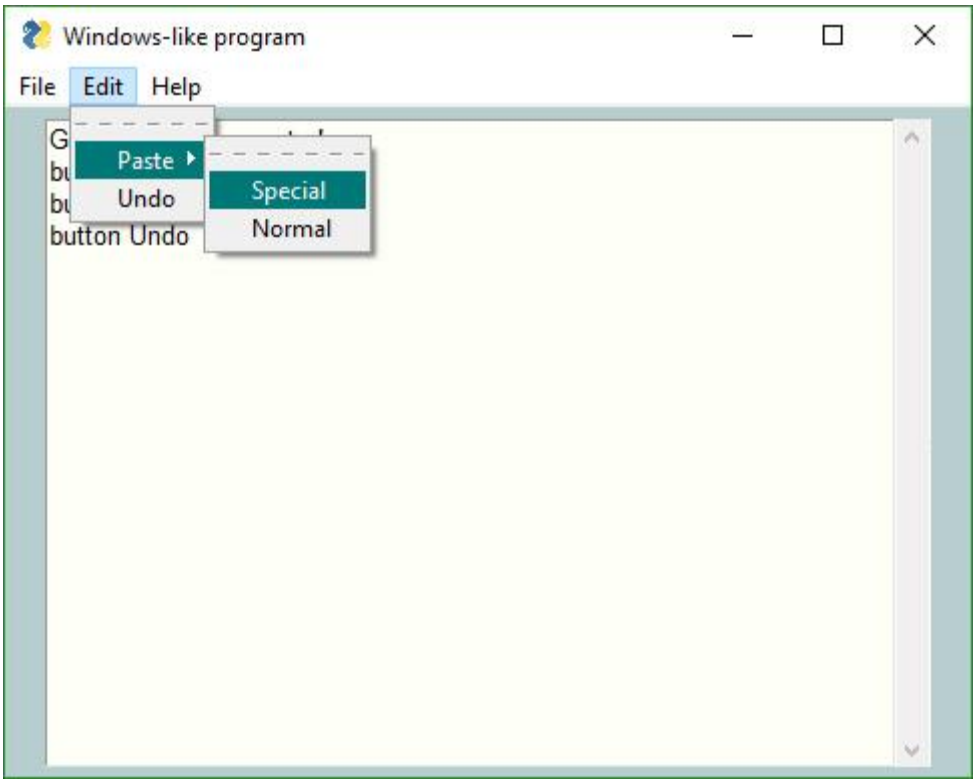
Beginning in version 3.01 you can add a menubar to your window. You specify the menus in much the same way as you do window layouts, with lists. Menu selections are returned as button clicks, so be aware of your overall naming conventions. If you have an Exit button and also an Exit menu option, then you won't be able to tell the difference when your window.Read returns. Hopefully will not be a problem.

This definition:

```
menu_def = [['File', ['Open', 'Save', 'Exit',]],
            ['Edit', ['Paste', ['Special', 'Normal',], 'Undo'],],
            ['Help', 'About...'],]
```

Note the placement of ',' and of []. It's tricky to get the nested menus correct that implement cascading menus. See how paste has Special and Normal as a list after it. This means that Paste has a cascading menu with items Special and Normal.

They menu_def layout produced this window:



You have used ALT-key in other Windows programs to navigate menus. For example Alt-F+X exits the program. The Alt-F pulls down the File menu. The X selects the entry marked Exit.

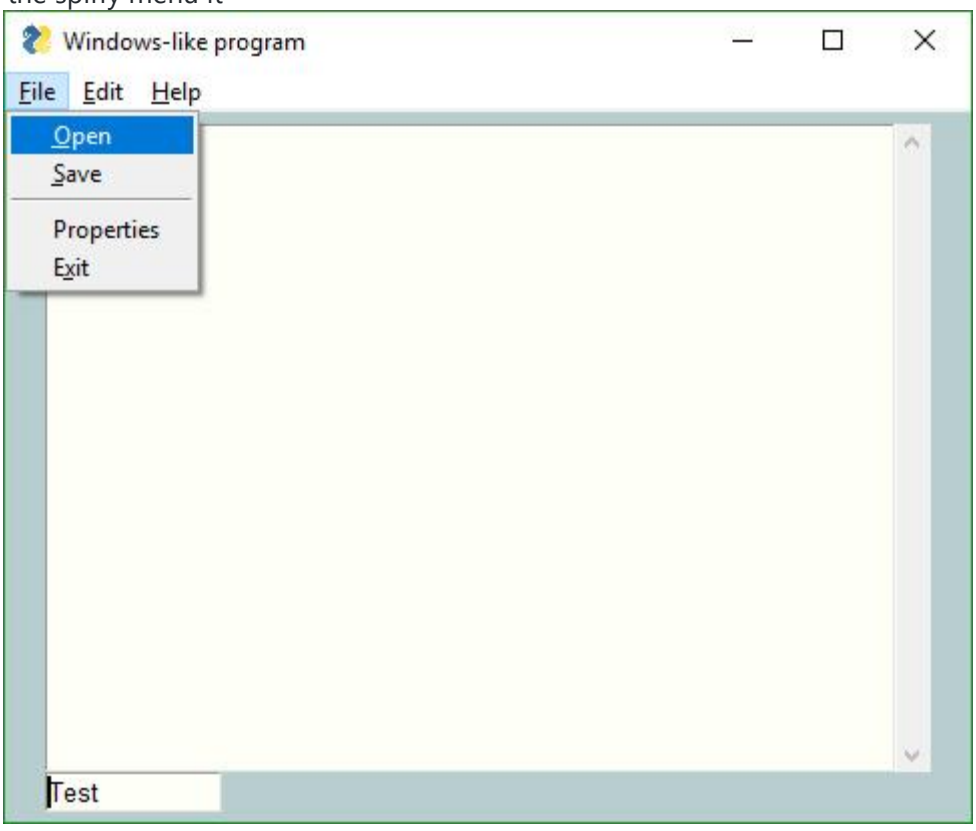
The good news is that PySimpleGUI allows you to create the same kind of menus! Your program can play with the big-boys. And, it's trivial to do.

All that's required is for your to add an "&" in front of the letter you want to appear with an underscore. When you hold the Alt key down you will see the menu with underlines that you marked.

One other little bit of polish you can add are separators in your list. To add a line in your list of menu choices, create a menu entry that looks like this: `'---'`
This is an example Menu with underlines and a separator.

```
# ------ Menu Definition ------ #
menu_def = [['&File', ['&Open', '&Save', '---', 'Properties', 'E&xit'  ]],
            ['&Edit', ['Paste', ['Special', 'Normal',], 'Undo'],],
            ['&Help', '&About...'],]
```

And this is the spiffy menu it produced:



Updating Elements

This is a somewhat advanced topic...

Typically you perform Element updates in response to events from other Elements. An example is that when you click a button some text on the window changes to red. You can change the Element's attributes, or at least some of them, and the Element's value.

In some source code examples you will find an older techique for updating elements that did not involve keys. If you see a technique in the code that does not use keys, then know that there is a version using keys that is easier.

Here's the key's version.... We have an InputText field that we want to update. When the Element was created we used this call:

```
sg.Input(key='input')
```

To update or change the value for that Input Element, we use this construct:

```
window.FindElement('input').Update('new text')
```

Using the '.' makes the code shorter. The FindElement call returns an Element. We then call that Element's Update function.

See the Font Sizer demo for example source code.

You can use Update to do things like:

- Have one Element (appear to) make a change to another Element
- Disable a button, slider, input field, etc
- Change a button's text
- Change an Element's text or background color
- Add text to a scrolling output window
- Change the choices in a list
- etc

Updating Multiple Elements

If you have a large number of Elements to update, you can call `Window.UpdateElements()`.

```
UpdateElements(key_list, value_list)
```

`key_list` - list of keys for elements you wish to update `value_list` - list of values, one for each key

```
window.UpdateElements(('name', 'address', 'phone'), ('Fred Flintstone', '123 Rock Quarry Road', '555#'))
```

Sample Applications

Use the example programs as a starting basis for your GUI. Copy, paste, modify and run! The demo files are:

Source File	Description
Demo_All_Widgets.py	Nearly all of the Elements shown in a single window
Demo_Borderless_Window.py	Create clean looking windows with no border
Demo_Button_States.py	One way of implementing disabling of buttons
Demo_Calendar.py	Demo of the Calendar Chooser button
Demo_Canvas.py	window with a Canvas Element that is updated outside of the window
Demo_Chat.py	A chat window with scrollable history
Demo_Chatterbot.py	Front-end to Chatterbot Machine Learning project
Demo_Color.py	How to interact with color using RGB hex values and named colors
Demo_Columns.py	Using the Column Element to create more complex windows
Demo_Compare_Files.py	Using a simple GUI front-end to create a compare 2-files utility
Demo_Cookbook_Browser.py	Source code browser for all Recipes in Cookbook
Demo_Dictionary.py	Specifying and using return values in dictionary format
Demo_DOC_Viewer_PIL.py	Display a PDF, HTML, ebook file, etc in your window
Demo_DisplayHash1and256.py	Using high level API and custom window to implement a simple display hash code utility

Source File	Description
Demo_DuplicateFileFinder.py	High level API used to get a folder that is used by utility that finds duplicate files. Uses progress meter to show progress. 2 lines of code required to add GUI and meter
Demo_Fill_Form.py	How to perform a bulk-fill for a window. Saving and loading a window from disk
Demo Font Sizer.py	Demonstrates Elements updating other Elements
Demo_Func_Callback_Simulator.py	For the Raspberry Pi crowd. Event loop that simulates traditional GUI callback functions should you already have an architecture that uses them
Demo_GoodColors.py	Using some of the pre-defined PySimpleGUI individual colors
Demo_HowDoI.py	This is a utility to be experienced! It will change how you code
Demo_Img_Viewer.py	Display jpg, png,tiff, bmp files
Demo_Keyboard.py	Using blocking keyboard events
Demo_Keyboard_Realtime.py	Using non-blocking / realtime keyboard events
Demo_Machine_Learning.py	A sample Machine Learning front end
Demo_Matplotlib.py	Integrating with Matplotlib to create a single graph
Demo_Matplotlib_Animated.py	Animated Matplotlib line graph
Demo_Matplotlib_Animated_Scatter.py	Animated Matplotlib scatter graph

Source File	Description
Demo_Matplotlib_Browser.py	Browse Matplotlib gallery
Demo_Media_Player.py	Non-blocking window with a media player layout. Demonstrates button graphics, Update method
Demo_MIDI_Player.py	GUI wrapper for Mido MIDI package. Functional MIDI player that controls attached MIDI devices
Demo_NonBlocking_Form.py	a basic async window
Demo_OpenCV.py	Integrated with OpenCV
Demo_Password_Login	Password protection using SHA1
Demo_PDF_Viewer.py	Submitted by a user! Previews PDF documents. Uses keyboard input & mouse scrollwheel to navigate
Demo_Pi_LEDs.py	Control GPIO using buttons
Demo_Pi_Robotics.py	Simulated robot control using realtime buttons
Demo_PNG_Vierwer.py	Uses Image Element to display PNG files
Demo_Progress_Meters.py	Demonstrates using 2 progress meters simultaneously
Demo_Recipes.py	A collection of various Recipes. Note these are not the same as the Recipes in the Recipe Cookbook
Demo_Script_Launcher.py	Demonstrates one way of adding a front-end onto several command line scripts

Source File	Description
Demo_Script_Parameters.py	Add a 1-line GUI to the front of your previously command-line only scripts
Demo_Tabbed_Form.py	Using the Tab feature
Demo_Table_Simulation.py	Use input fields to display and edit tables
Demo_Timer.py	Simple non-blocking window

Packages Used In Demos

While the core PySimpleGUI code does not utilize any 3rd party packages, some of the demos do. They add a GUI to a few popular packages. These packages include:

- Chatterbot
- Mido
- Matplotlib
- PyMuPDF

Creating a Windows .EXE File

It's 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.

Installation of the packages, you'll need to install PySimpleGUI and PyInstaller (you need to install only once)

```
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.

If you get a crash with something like:

```
ValueError: script '.......\src\tkinter' not found
```
Then try adding **`--hidden-import tkinter`** to your command

Fun Stuff

Here are some things to try if you're bored or want to further customize

Debug Output Be sure and check out the EasyPrint (Print) function described in the high-level API section. Leave your code the way it is, route your stdout and stderror to a scrolling window.

For a fun time, add these lines to the top of your script

```
import PySimpleGUI as sg
print = sg.Print
```
This will turn all of your print statements into prints that display in a window on your screen rather than to the terminal.

Look and Feel Dial in the look and feel that you like with the `SetOptions` function. You can change all of the defaults in one function call. One line of code to customize the entire GUI. Or beginning in version 2.9 you can choose from a look and feel using pre-defined color schemes. Call ChangeLookAndFeel with a description string.
```
sg.ChangeLookAndFeel('GreenTan')
```
Valid values for the description string are:

```
  GreenTan
  LightGreen
  BluePurple
  Purple
  BlueMono
  GreenMono
  BrownBlue
  BrightColors
  NeutralBlue
  Kayak
  SandyBeach
  TealMono
```
To see the latest list of color choices, take a look at the bottom of the `PySimpleGUI.py` file where you'll find the `ChangLookAndFeel` function.
You can also combine the `ChangeLookAndFeel` function with the `SetOptions` function to quickly modify one of the canned color schemes. Maybe you like the colors but was more depth to your bezels. You can dial in exactly what you want.
ObjToString Ever wanted to easily display an objects contents easily? Use ObjToString to get a nicely formatted recursive walk of your objects. This statement:

```
print(sg.ObjToSting(x))
```
And this was the output

```
<class '__main__.X'>
    abc = abc
    attr12 = 12
    c = <class '__main__.C'>
        b = <class '__main__.B'>
            a = <class '__main__.A'>
                attr1 = 1
                attr2 = 2
                attr3 = three
            attr10 = 10
            attrx = x
```

You'll quickly wonder how you ever coded without it.

Known Issues

While not an "issue" this is a ***stern warning***

Do not attempt to call `PySimpleGUI` from multiple threads! It's `tkinter` based and `tkinter` has issues with multiple threads

Progress Meters - the visual graphic portion of the meter may be off. May return to the native tkinter progress meter solution in the future. Right now a "custom" progress meter is used. On the bright side, the statistics shown are extremely accurate and can tell you something about the performance of your code. If you are running 2 or more progress meters at the same time using `OneLineProgressMeter`, you need to close the meter by using the "Cancel" button rather than the X

Async windows - these include the 'easy' windows (`OneLineProgressMeter` and EasyPrint/Print). If you start overlapping having Async windows open with normal windows then things get a littler squirrelly. Still tracking down the issues and am making it more solid every day possible. You'll know there's an issue when you see blank window.

EasyPrint - EasyPrint is a new feature that's pretty awesome. You print and the output goes to a window, with a scroll bar, that you can copy and paste from. Being a new feature, it's got some potential problems. There are known interaction problems with other GUI windows. For example, closing a Print window can also close other windows you have open. For now, don't close your debug print window until other windows are closed too.

Contributing

A MikeTheWatchGuy production... entirely responsible for this code.... unless it causes you trouble in which case I'm not at all responsible.

Versions

Version	Description
1.0.9	July 10, 2018 - Initial Release
1.0.21	July 13, 2018 - Readme updates
2.0.0	July 16, 2018 - ALL optional parameters renamed from CamelCase to all_lower_case
2.1.1	July 18, 2018 - Global settings exposed, fixes
2.2.0	July 20, 2018 - Image Elements, Print output
2.3.0	July 23, 2018 - Changed form.Read return codes, Slider Elements, Listbox element. Renamed some methods but left legacy calls in place for now.
2.4.0	July 24, 2018 - Button images. Fixes so can run on Raspberry Pi
2.5.0	July 26, 2018 - Colors. Listbox scrollbar. tkinter Progress Bar instead of homegrown.
2.6.0	July 27, 2018 - auto_size_button setting. License changed to LGPL 3+
2.7.0	July 30, 2018 - realtime buttons, window_location default setting
2.8.0	Aug 9, 2018 - New None default option for Checkbox element, text color option for all elements, return values as a dictionary, setting focus, binding return key
2.9.0	Aug 16,2018 - Screen flash fix, `do_not_clear` input field option, `autosize_text` defaults to `True` now, return values as ordered dict, removed text target from progress bar, rework of return values and initial return values, removed legacy Form.Refresh() method (replaced by Form.ReadNonBlockingForm()), COLUMN elements!!, colored text defaults
2.10.0	Aug 25, 2018 - Keyboard & Mouse features (Return individual keys as if buttons, return mouse scroll-wheel as button, bind return-key to button, control over keyboard focus), SaveAs Button, Update & Get methods for InputText, Update for Listbox, Update & Get for Checkbox, Get for Multiline, Color options for Text Element Update, Progess bar Update can change max value, Update for Button to change text & colors, Update for Image Element, Update for Slider, Form level text justification, Turn off default focus, scroll bar for Listboxes, Images can be from filename or from in-RAM, Update for Image). Fixes - text wrapping in buttons, msg box, removed slider borders entirely and others

Version	Description
2.11.0	Aug 29, 2018 - Lots of little changes that are needed for the demo programs to work. Buttons have their own default element size, fix for Mac default button color, padding support for all elements, option to immediately return if list box gets selected, FilesBrowse button, Canvas Element, Frame Element, Slider resolution option, Form.Refresh method, better text wrapping, 'SystemDefault' look and feel settin
2.20.0	Sept 4, 2018 - Some sizable features this time around of interest to advanced users. Renaming of the MsgBox functions to Popup. Renaming GetFile, etc, to PopupGetFile. High-level windowing capabilities start with Popup, PopupNoWait/PopupNonblocking, PopupNoButtons, default icon, change_submits option for Listbox/Combobox/Slider/Spin/, New OptionMenu element, updating elements after shown, system defaul color option for progress bars, new button type (Dummy Button) that only closes a window, SCROLLABLE Columns!! (yea, playing in the Big League now), LayoutAndShow function removed, form.Fill - bulk updates to forms, FindElement - find element based on key value (ALL elements have keys now), no longer use grid packing for row elements (a potentially huge change), scrolled text box sizing changed, new look and feel themes (Dark, Dark2, Black, Tan, TanBlue, DarkTanBlue, DarkAmber, DarkBlue, Reds, Green)
2.30.0	Sept 6, 2018 - Calendar Chooser (button), borderless windows, load/save form to disk
3.0.0	Sept 7, 2018 - The "fix for poor choice of 2.x numbers" release. Color Chooser (button), "grab anywhere" windows are on by default, disable combo boxes, Input Element text justification (last part needed for 'tables'), Image Element changes to support OpenCV?, PopupGetFile and PopupGetFolder have better no_window option
3.01.01	Sept 10, 2018 - Menus! (sort of a big deal)
3.01.02	Step 11, 2018 - All Element.Update functions have a `disabled` parameter so they can be disabled. Renamed some parameters in Update function (sorry if I broke your code), fix for bug in Image.Update. Wasn't setting size correctly, changed grab_anywhere logic again,added grab anywhere option to PupupGetText (assumes disabled)
3.02.00	Sept 14, 2018 - New Table Element (Beta release), MsgBox removed entirely, font setting for InputText Element, **packing change** risky change that allows some Elements to be resized,removed command parameter from Menu Element, new function names for ReadNonBlocking (Finalize, PreRead), change to text element autosizing and wrapping (yet again), lots of parameter additions to Popup functions (colors, etc).
3.03.00	New feature - One Line Progress Meters, new display_row_numbers for Table Element, fixed bug in EasyProgresssMeters (function will soon go away), OneLine and Easy progress meters set to grab anywhere but can be turned off.
03,04.00	Sept 18, 2018 - New features - Graph Element, Frame Element, more settings exposed to Popup calls. See notes below for more.

Version	Description
03.04.01	Sept 18, 2018 - See release notes
03.05.00	Sept 20, 2018 - See release notes
03.05.01	Sept 22, 2018 - See release notes
03.05.02	Sept 23, 2018 - See release notes
03.06.00	Sept 23, 2018 - Goodbye FlexForm, hello Window
03.08.00	Sept 25, 2018 - Tab and TabGroup Elements\
01.01.00 for 2.7	Sept 25, 2018 - First release for 2.7

Release Notes

2.3 - Sliders, Listbox's and Image elements (oh my!)

If using Progress Meters, avoid cancelling them when you have another window open. It could lead to future windows being blank. It's being worked on.

New debug printing capability. `sg.Print`
2.5 Discovered issue with scroll bar on `Output` elements. The bar will match size of ROW not the size of the element. Normally you never notice this due to where on a form the `Output` element goes. Listboxes are still without scrollwheels. The mouse can drag to see more items. The mouse scrollwheel will also scroll the list and will `page up` and `page down` keys.
2.7 Is the "feature complete" release. Pretty much all features are done and in the code

2.8 More text color controls. The caller has more control over things like the focus and what buttons should be clicked when enter key is pressed. Return values as a dictionary! (NICE addition)

2.9 COLUMNS! This is the biggest feature and had the biggest impact on the code base. It was a difficult feature to add, but it was worth it. Can now make even more layouts. Almost any layout is possible with this addition.

..... insert releases 2.9 to 2.30 .....

3.0 We've come a long way baby! Time for a major revision bump. One reason is that the numbers started to confuse people the latest release was 2.30, but some people read it as 2.3 and thought it went backwards. I kinda messed up the 2.x series of numbers, so why not start with a clean slate. A lot has happened anyway so it's well earned.

One change that will set PySimpleGUI apart is the parlor trick of being able to move the window by clicking on it anywhere. This is turned on by default. It's not a common way to interact with windows. Normally you have to move using the titlebar. Not so with PySimpleGUI. Now you can drag using any part of the window. You will want to turn this off for windows with sliders. This feature is enabled in the Window call.

Related to the Grab Anywhere feature is the no_titlebar option, again found in the call to Window. Your window will be a spiffy, borderless window. It's a really interesting effect. Slight problem is that you do not have an icon on the taskbar with these types of windows, so if you don't supply a button to close the window, there's no way to close it other than task manager.

3.0.2 Still making changes to Update methods with many more ahead in the future. Continue to mess with grab anywhere option. Needed to disable in more places such as the PopupGetText function. Any time these is text input on a form, you generally want to turn off the grab anywhere feature.

3.2.0

Biggest change was the addition of the Table Element. Trying to make changes so that form resizing is a possibility but unknown if will work in the long run. Removed all MsgBox, Get* functions and replaced with Popup functions. Popups had multiple new parameters added to change the look and feel of a popup.

3.3.0

OneLineProgressMeter function added which gives you not only a one-line solution to progress meters, but it also gives you the ability to have more than 1 running at the same time, something not possible with the EasyProgressMeterCall

3.4.0

- Frame - New Element - a labelled frame for grouping elements. Similar to Column
- Graph (like a Canvas element except uses the caller's coordinate system rather than tkinter's).
- initial_folder - sets starting folder for browsing type buttons (browse for file/folder).
- Buttons return key value rather than button text **If** a key is specified,
- OneLineProgressMeter! Replaced EasyProgressMeter (sorry folks that's the way progress works sometimes)
- Popup - changed ALL of the Popup calls to provide many more customization settings
    - Popup
    - PopupGetFolder
    - PopupGetFile
    - PopupGetText
    - Popup
    - PopupNoButtons
    - PopupNonBlocking
    - PopupNoTitlebar
    - PopupAutoClose
    - PopupCancel
    - PopupOK
    - PopupOKCancel
    - PopupYesNo

3.4.1

- Button.GetText - Button class method. Returns the current text being shown on a button.
- Menu - Tearoff option. Determines if menus should allow them to be torn off
- Help - Shorcut button. Like Submit, cancel, etc
- ReadButton - shortcut for ReadFormButton

3.5.0

- Tool Tips for all elements
- Clickable text
- Text Element relief setting
- Keys as targets for buttons
- New names for buttons:
    - Button = SimpleButton
    - RButton = ReadButton = ReadFormButton
- Double clickable list entries
- Auto sizing table widths works now
- Feature DELETED - Scaling. Removed from all elements

3.5.1

- Bug fix for broken PySimpleGUI if Python version < 3.6 (sorry!)
- LOTS of Readme changes

3.5.2

- Made `Finalize()` in a way that it can be chained
- Fixed bug in return values from Frame Element contents

3.6.0

- Renamed FlexForm to Window
- Removed LookAndFeel capability from Mac platform.

3.8.0

- Tab and TabGroup Elements - awesome new capabilities

1.0.0 Python 2.7

It's official. There is a 2.7 version of PySimpleGUI!

3.8.2

- Exposed `TKOut` in Output Element
- `DrawText` added to Graph Elements
- Removed `Window.UpdateElements`
- `Window.grab_anywere` defaults to False

3.8.3

- Listbox, Slider, Combobox, Checkbox, Spin, Tab Group - if change_submits is set, will return the Element's key rather than ''
- Added change_submits capability to Checkbox, Tab Group
- Combobox - Can set value to an Index into the Values table rather than the Value itself
- Warnings added to Drawing routines for Graph element (rather than crashing)
- Window - can "force top level" window to be used rather than a normal window. Means that instead of calling Tk to get a window, will call TopLevel to get the window
- Window Disable / Enable - Disables events (button clicks, etc) for a Window. Use this when you open a second window and want to disable the first window from doing anything. This will simulate a 'dialog box'
- Tab Group returns a value with Window is Read. Return value is the string of the selected tab
- Turned off grab_anywhere for Popups
- New parameter, default_extension, for PopupGetFile
- Keyboard shortcuts for menu items. Can hold ALT key to select items in men
- Removed old-style Tabs - Risky change because it hit fundamental window packing and creation. Will also break any old code using this style tab (sorry folks this is how progress happens)

3.8.3

- Fix for Menus.
- Fixed tabled colors. Now they work
- Fixed returning keys for tabs
-

Upcoming

Make suggestions people! Future release features

Port to other graphic engines. Hook up the front-end interface to a backend other than tkinter. Qt, WxPython, etc. WxPython is higher priority.

Code Condition

```
Make it run
Make it right
Make it fast
```

It's a recipe for success if done right. PySimpleGUI has completed the "Make it run" phase. It's far from "right" in many ways. These are being worked on. The module is particularly poor for PEP 8 compliance. It was a learning exercise that turned into a somewhat complete GUI solution for lightweight problems.

While the internals to PySimpleGUI are a tad sketchy, the public interfaces into the SDK are more strictly defined and comply with PEP 8 for the most part.

Please log bugs and suggestions in the GitHub! It will only make the code stronger and better in the end, a good thing for us all, right?

Design

A moment about the design-spirit of `PySimpleGUI`. From the beginning, this package was meant to take advantage of Python's capabilities with the goal of programming ease.
Single File While not the best programming practice, the implementation resulted in a single file solution. Only one file is needed, PySimpleGUI.py. You can post this file, email it, and easily import it using one statement.

Functions as objects In Python, functions behave just like object. When you're placing a Text Element into your form, you may be sometimes calling a function and other times declaring an object. If you use the word Text, then you're getting an object. If you're using `Txt`, then you're calling a function that returns a `Text` object.
Lists It seemed quite natural to use Python's powerful list constructs when possible. The form is specified as a series of lists. Each "row" of the GUI is represented as a list of Elements. When the form read returns the results to the user, all of the results are presented as a single list. This makes reading a form's values super-simple to do in a single line of Python code.

Dictionaries Want to view your form's results as a dictionary instead of a list... no problem, just use the `key` keyword on your elements. For complex forms with a lot of values that need to be changed frequently, this is by far the best way of consuming the results.
You can also look up elements using their keys. This is an excellent way to update elements in reaction to another element. Call `form.FindElement(key)` to get the Element.

Author

MikeTheWatchGuy

Demo Code Contributors

JorjMcKie - PDF and image viewers (plus a number of code suggestions) Otherion - Table Demos Panda & CSV. Loads of suggestions to the core APIs

License

GNU Lesser General Public License (LGPL 3) +

Acknowledgments

- JorjMcKie was the motivator behind the entire project. His wxsimpleGUI concepts sparked PySimpleGUI into existence
- Fredrik Lundh for his work on `tkinter`
- Ruud van der Ham for all the help he's provided as a Python-mentor. Quite a few tricky bits of logic was supplied by Ruud. The dual-purpose return values scheme is Ruud's for example
- **Numerous** users who provided feature suggestions! Many of the cool features were suggested by others. If you were one of them and are willing to take more credit, I'll list you here if you give me permission. Most are too modest
- moshekaplan/**tkinter_components** wrote the code for the Calendar Chooser Element. It was lifted straight from GitHub
- Bryan Oakley for the code that enables the `grab_anywhere` feature.
- Otherion for help with Tables, being a sounding board for new features, naming functions, ..., all around great help
- agjunyent figured out how to properly make tabs and wrote prototype code that demonstrated how to do it
- jfongattw huge suggestion... dictionaries. turned out to be
- one of the most critical constructs in PySimpleGUI
- venim code to doing Alt-Selections in menus, updating Combobox using index, request to disable windows (a really good idea), checkbox and tab submits on change, returning keys for elements that have change_submits set, ...

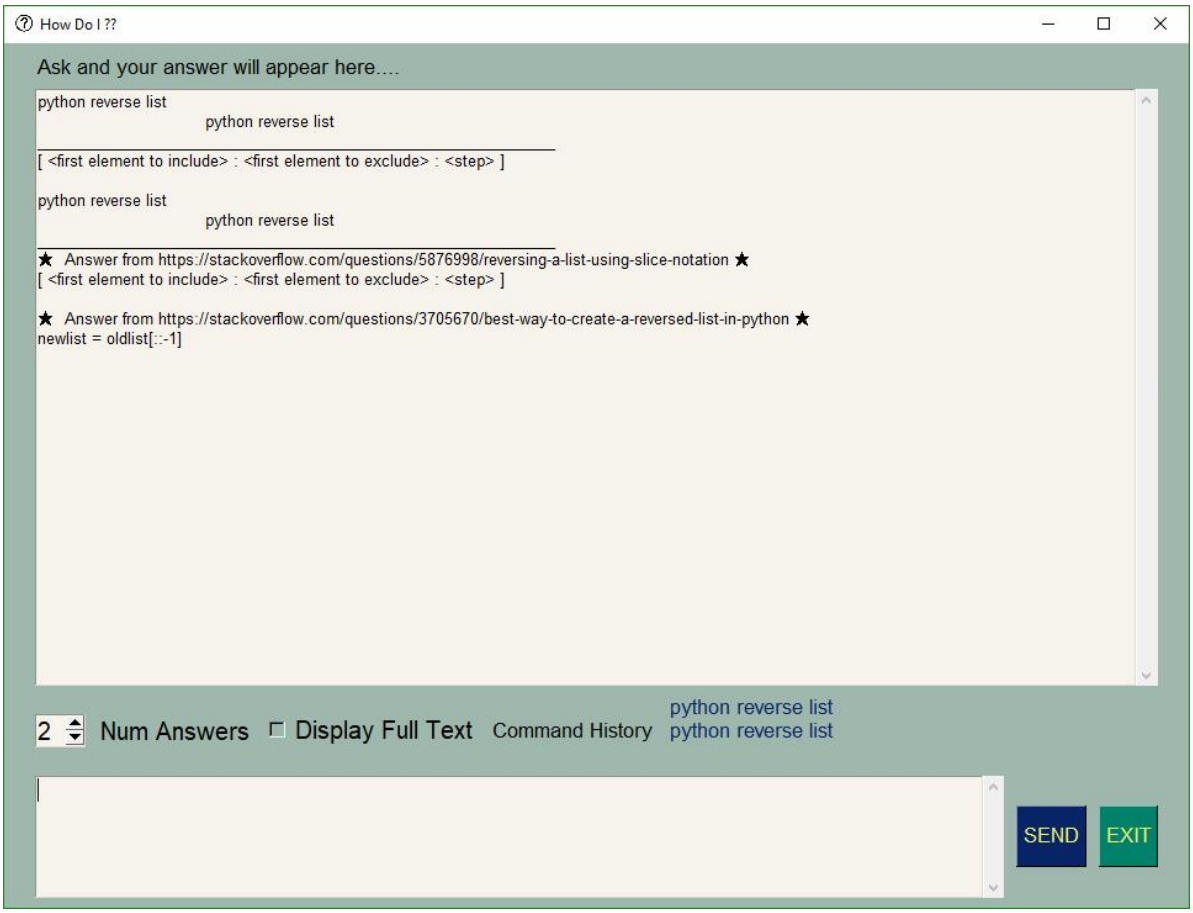
How Do I

Finally, I must thank the fine folks at How Do I. https://github.com/gleitz/howdoi Their utility has forever changed the way and pace in which I can program. I urge you to try the HowDoI.py application here on GitHub. Trust me, **it's going to be worth the effort!** Here are the steps to run that application

```
Install howdoi:
      pip install howdoi
Test your install:
      python -m howdoi howdoi.py
To run it:
      Python HowDoI.py
```

The pip command is all there is to the setup.

The way HowDoI works is that it uses your search term to look through stack overflow posts. It finds the best answer, gets the code from the answer, and presents it as a response. It gives you the correct answer OFTEN. It's a miracle that it work SO well. For Python questions, I simply start my query with 'Python'. Let's say you forgot how to reverse a list in Python. When you run HowDoI and ask this question, this is what you'll see.



In the hands of a competent programmer, this tool is **amazing**. It's a must-try kind of program that has completely changed my programming process. I'm not afraid of asking for help! You just have to be smart about using what you find.

The PySimpleGUI window that the results are shown in is an 'input' field which means you can copy and paste the results right into your code.