PySimpleGUI

tkinter tkinter 2.7 Qt WxPython Web (Remi)

docs passing

downloads 231k

downloads 105k

downloads 96k

downloads 30k

downloads 114k

Awesome meter 100

Python 2.7 3.x

PYSIMPLEGUI FOR PYTHON 3.X VERSION 4.5.0

PYSIMPLEGUI FOR PYTHON 2.7 VERSION 2.4.1

PYSIMPLEGUIQT VERSION 0.28.0

PYSIMPLEGUIWX VERSION 0.11.0

PYSIMPLEGUIWEB VERSION 0.28.1

PySimpleGUI User's Manual

Python GUI For Humans - Transforms tkinter, Qt, Remi, WxPython into portable people-friendly Pythonic interfaces

This manual is crammed full of answers so start your search for answers here. Read/Search this prior to opening an Issue on GitHub. Press Control F and type.

Jump-Start

Install

pip install pysimplegui
or
pip3 install pysimplegui

This Code

```
import PySimpleGUI as sg
sg.change_look_and_feel('DarkAmber')

layout = [  [sg.Text('Some text on Row 1')],
            [sg.Text('Enter something on Row 2'), sg.InputText()],
            [sg.Button('Ok'), sg.Button('Cancel')] ]

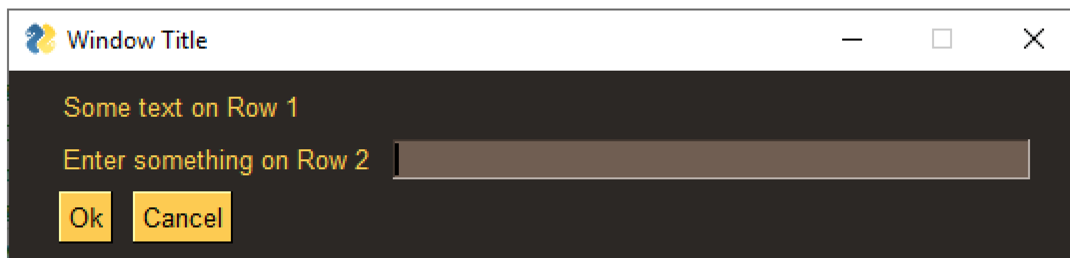

window = sg.Window('Window Title', layout)

while True:
    event, values = window.read()
    if event in (None, 'Cancel'):
        break
    print('You entered ', values[0])

window.close()
```

Makes This Window

and returns the value input as well as the button clicked.



Any Questions? It's that simple.

Looking for a GUI package? Are you....

- looking to take your Python code from the world of command lines and into the convenience of a GUI?
- sitting on 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?
- an IT guy/gal that has written some cool tools but due to corporate policies are unable to share unless an EXE file?
- wanting to distribute your Python code to Windows users as a single .EXE file that launches straight into a GUI, much like a WinForms app?
- want to share your program with your friends or families (that aren't so freakish that they have Python running)
- wanting to run a program in your system tray?
- a teacher wanting to teach your students how to program using a GUI?

- a student that wants to put a GUI onto your project that will blow away your teacher?
- looking for a GUI package that is "supported" and is being constantly developed to improve it?
- longing for documentation and scores of examples?

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

The basics

- Create windows that look and operate *identically* to those created directly with tkinter, Qt, WxPython, and Remi.
- Requires 1/2 to 1/10th the amount of code as underlying frameworks.
- One afternoon is all that is required to learn the PySimpleGUI package *and* write your first custom GUI.
- Students can begin using within their first week of Python education.
- No callback functions. You do not need to write the word `class` *anywhere* in your code.
- Access to nearly every underlying GUI Framework's Widgets.
- Supports both Python 2.7 & 3 when using tkinter
- Supports both PySide2 and PyQt5 (limited support)
- Effortlessly move across tkinter, Qt, WxPython, and the Web (Remi) by changing only the import statement
- The *only* way to write both desktop and web based GUIs at the same time in Python
- Developed from nothing as a pure Python impelementation with Python friendly interfaces.
- Run your program in the System Tray using WxPython. Or, change the import and run it on Qt with no other changes.
- Works with Qt Designer
- Built in Debugger
- Actively maintained and enhanced - 4 ports are underway, all being used by users.
- Corporate as well as home users.
- Appealing to both newcomers to Python and experiened Pythonistas.
- The focus is entirely on the developer (you) and making their life easier, simplified, and in control.
- 170+ Demo Programs teach you how to integrate with many popular packages like OpenCV, Matplotlib, PyGame, etc.
- 200 pages of documentation, a Cookbook, built-in help using docstrings, in short it's heavily documented

July-2019 Note - This readme is being generated from the PySimpleGUI.py file located on GitHub. As a result, some of the calls or parameters may not match the PySimpleGUI that you pip installed.

GUI Development does not have to be difficult nor painful. It can be FUN

What users are saying about PySimpleGUI

(None of these comments were solicited & are not paid endorsements - other than a huge thank you they received!)

"I've been working to learn PyQT for the past week in my off time as an intro to GUI design and how to apply it to my existing scripts... Took me ~30 minutes to figure out PySimpleGUI and get my scripts working with a GUI."

"Python has been an absolute nightmare for me and I've avoided it like the plague. Until I saw PysimpleGUI."

"I've been pretty amazed at how much more intuitive it is than raw tk/qt. The dude developing it is super active on the project too so if you come across situations that you just can't get the code to do what you want you can make bug/enhancement issues that are almost assured to get a meaningful response."

"This library is the easiest way of gui programming in python! I'm totally in love with it"

"Wow that readme is extensive and great." (hear the love for docs often)

"Coming from R, Python is absolutely slick for GUIs. PySimpleGUI is a dream."

"I have been writing Python programs for about 4 or 5 months now. Up until this week I never had luck with any UI libraries like Tkinter, Qt, Kivy. I went from not even being able to load a window in Tkinter reliably to making a loading screen, and full program in one night with PySimpleGUI."

"I love PySimpleGUI! I've been teaching it in my Python classes instead of Tkinter."

"I wish PySimpleGUI was available for every friggin programming language"

START HERE - User Manual with Table of Contents

ReadTheDocs <------ THE best place to read the docs due to TOC, all docs in 1 place, and better formatting. START here in your education. Easy to remember PySimpleGUI.org.

Quick Links To Help and The Latest News and Releases

Homepage - Lastest Readme and Code - GitHub Easy to remember: PySimpleGUI.com

Announcements of Latest Developments, Release news, Misc

COOKBOOK!

Brief Tutorial

Latest Demos and Master Branch on GitHub

Repl.it Home for PySimpleGUI

Lots of screenshots

How to submit an Issue

The YouTube videos - If you like instructional videos, there are over 15 videos - 5 part series of basics - 10 part series of more detail - The Naked Truth (An updaate on the technology) - There are numerous short videos also on that channel that demonstrate PySimpleGUI being used

About The PySimpleGUI Documentation System

This User's Manual (also the project's readme) is one ***vital*** part of the PySimpleGUI programming environment.

If you are a professional or skilled in how to develop software, then you understand the role of documentation in the world of technology development. You can skip this bit.... look for the bold "**GO TO HERE**" below.

RTFM is not a new acronym. It stretches back to 1979, the dawn of the computer-era and in particular the microprocessor. The point is that this is not a new problem. It's a very old problem.

Bluntness is required here as the subtle approach has not worked in the past:

It WILL be required, at times, for you to read or search this document in order to be successful.

Re-read that statement. This **will** be a serious problem for you if you're the type of person that finds it "quicker and easier to post on StackOverflow rather than reading documentation".

If you have not yet matured to the point you are able to understand this skill or choose to not follow it, then please save ***everyone*** the pain of doing **for you** what you, as a developer, software engineer, or wanna be coder, must do on your own. It's a vital skill for you to learn.

Want to be a "real engineer"? Then follow "real engineering practices" such as "reading". You are learning a NEW GUI package. You've not seen anything like it. Don't be so arrogant as to believe you will never need to read documentation.

UGH, why does this need to be said?

GO TO HERE if instructed above.

I appologize to the other 95% of you that this..... pathetic.... reminder needs to be added, but sadly there's a need for it.

There are 5 resources that work together to provide to you the fastest path to success. They are:

1. This User's Manual
2. The Cookbook
3. The 170+ Demo Programs
4. Docstrings enable you to access help directly from Python or your IDE
5. Searching the GitHub Issues as a last resort (search both open and closed issues)

Pace yourself. The intial progress is exciting and FAST PACED. However, GUIs take time and thought to build. Take a deep breath and use the provided materials and you'll do fine. Don't skip the design phase of your GUI after you run some demos and get the hang of things. If you've tried other GUI frameworks before, successful or not, then you know you're already way ahead of the game using PySimpleGUI versus the underlying GUI frameworks. It may feel like the 3 days you've been working on your code has been forever, but by comparison of 3 days learning Qt, PySimpleGUI will look trivial to learn.

It is not by accident that this section, about documentation, is at the TOP of this document.

This documentation is not HUGE in length for a package this size. In fact it's still one document and it's the readme for the GitHub. It's not written in complex english. It is understandable by complete beginners. And pressing `Control+F` is all you need to do to search this document. USUALLY you'll find less than 6 matches.

Documentation and Demos Get Out of Date

Sometimes the documenation doesn't match exactly the version of the code you're running. Sometimes demo programs haven't been updated to match a change made to the SDK. Things don't happen simultaneously generally speaking. So, it may very well be that you find an error or inconsistency or something no longer works with the latest version of an external library.

If you've found one of these problems, and you've searched to make sure it's not a simple mistake on your part, then by ALL means log an Issue on the GitHub. Don't be afraid to report problems if you've taken the simple steps of checking out the docs first.

Platforms

Hardware and OS Support

PySimpleGUI runs on Windows, Linux and Mac, just like tkinter, Qt, WxPython and Remi do. If you can get the underlying GUI Framework installed / running on your machine then PySimpleGUI will also run there.

Hardware

- PC's, Desktop, Laptops
- Macs of all types
- Raspberry Pi
- Android devices like phones and tablets
- Virtual machine online (no hardware) - repl.it

OS

- Windows 7, 8, 10
- Linux on PC - Tested on **many** distributions
- Linux on Raspbnerry Pi
- Linux on Android - Can use either Termux or PyDroid3
- Mac OS (Sorry don't know much about Macs other than Macs don't like tkinter)

Python versions

As of 9/25/2018 **both Python 3 and Python 2.7 are supported** when using **tkinter version** of PySimpleGUI! 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. **None** of the other ports can use Python 2.

Python 2.7 Code will be deleted from this GitHub on Dec 31, 2019

Note that the 2.7 port will *cease to exist on this GitHub* on Jan 1, 2020. If you would like to know how much time you have to move over to the Python 3 version of PySimpleGUI, then go here: https://pythonclock.org/. The only thing that will be available is an unsupported PyPI release of PySimpleGUI27.

By "will cease to exist on this GitHub" I mean, it will be deleted entirely. No source code, no supporting programs. Nothing. If you're stuck using 2.7 in December, it would behoove you to fork the 2.7 code on Dec 31, 2019. Legacy Python doesn't have a permanent home here. It sounds cruel, but experts in security particularly says 2.7 is a huge risk. Furthering it use only hurts the computing world.

Warning - tkinter + Python 3.7.3 and later, including 3.8 has problems

The version of tkinter that is being supplied with the 3.7.3 and later versions of Python is known to have a problem with table colors. Basically, they don't work. As a result, if you want to use the plain PySimpleGUI running on tkinter, you should be using 3.7.2 or less. 3.6 is the

version PySimpleGUI has chosen as the recommended version for most users.

Output Devices

In addition to running as a desktop GUI, you can also run your GUI in a web browser by running PySimpleGUIWeb.

This is ideal for "headless" setups like a Raspberry Pi that is at the core of a robot or other design that does not have a normal display screen. For these devices, run a PySimpleGUIWeb program that never exits.

Then connect to your application by going to the Pi's IP address (and port #) using a browser and you'll be in communication with your application. You can use it to make configuration changes or even control a robot or other piece of hardward using buttons in your GUI

A Complete PySimpleGUI Program (Getting The Gist)

Before diving into details, here's a description of what PySimpleGUI is/does and why that is so powerful.

You keep hearing "custom window" in this document because that's what you're making and using... your own custom windows.

ELEMENTS is a word you'll see everywhere... in the code, documentation, ... Elements == PySimpleGUI's Widgets. As to not confuse a tkinter Button Widget with a PySimpleGUI Button Element, it was decided that PySimpleGUI's Widgets will be called Elements to avoid confusion.

Wouldn't it be nice if a GUI with 3 "rows" of Elements was defined in 3 lines of code? That's exactly how it's done. Each row of Elements is a list. Put all those lists together and you've got a window.

What about handling button clicks and stuff. That's 4 lines of the code below beginning with the while loop.

Now look at the `layout` variable and then look at the window graphic below. Defining a window is taking a design you can see visually and then visually creating it in code. One row of Elements = 1 line of code (can span more if your window is crowded). The window is exactly what we see in the code. A line of text, a line of text and an input area, and finally ok and cancel buttons.

This makes the coding process extremely quick and the amount of code very small

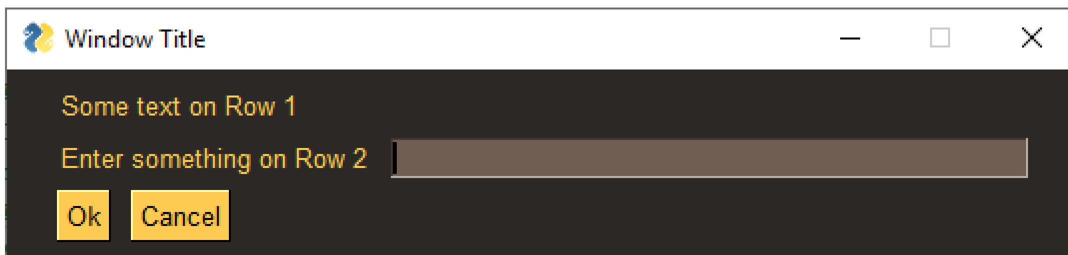
```
import PySimpleGUI as sg
sg.change_look_and_feel('DarkAmber')

layout = [  [sg.Text('Some text on Row 1')],
            [sg.Text('Enter something on Row 2'), sg.InputText()],
            [sg.OK(), sg.Cancel()]]


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

while True:
    event, values = window.read()
    if event in (None, 'Cancel'):
        break

window.close()
```



You gotta admit that the code above is a lot more "fun" looking that tkinter code you've studied before. Adding stuff to your GUI is ***trivial***. You can clearly see the "mapping" of those 3 lines of code to specific Elements laid out in a Window. It's not a trick. It's how easy it is to code in PySimpleGUI. With this simple concept comes the ability to create any window layout you wish. There are parameters to move elements around inside the window should you need more control.

It's a thrill to complete your GUI project way ahead of what you estimated. Some people take that extra time to polish their GUI to make it even nicer, adding more bells and whistles because it's so easy and it's a lot of fun to see success after success as you write your program.

Some are more advanced users and push the boundaries out and extend PySimpleGUI using their own extensions.

Others, like IT people and hackers are busily cranking out GUI program after GUI program, and creating tools that others can use. Finally there's an easy way to throw a GUI onto your program and give it to someone. It's a pretty big leap in capability for some people. It's GREAT to hear these successes. It's motivating for everyone in the end. Your success can easily motivate the next person to give it a try and also potentially be successful.

Usually there's a one to one mapping of a PySimpleGUI Element to a GUI Widget. A "Text Element" in PySimpleGUI == "Label Widget" in tkinter. What remains constant for you across all PySimpleGUI platforms is that no matter what the underlying GUI framework calls the thing

that places text in your window, you'll always use the PySimpleGUI Text Element to access it.

The final bit of magic is in how Elements are created and changed.

So far you've seen simply layouts with no customization of the Elements. Customizing and configuring Elements is another place PySimpleGUI utilizes the Python language to make your life easier.

What about Elements that have settings other than the standard system settings? What if I want my Text to be blue, with a Courier font on a green background. It's written quite simply:

```
Text('This is some text', font='Courier 12', text_color='blue', background_color='green')
```

The Python named parameters are ***extensively*** in PySimpleGUI. They are key in making the code compact, readable, and trivial to write.

As you'll learn in later sections that discuss the parameters to the Elements, there are a LOT of options avilable to you should you choose to use them. The `Text Element` has 15 parameters that you can change. This is one reason why PyCharm is suggested as your IDE... it does a fantastic job of displaying documentation as you type in your code.

That's *The Basics*

What do you think? Easier so far than your previous run-ins with GUIs in Python? Some programs, many in fact, are as simple as this example has been.

But PySimpleGUI certainly does **not** end here. This is the beginning. The scaffolding you'll build upon.

The Underlying GUI Frameworks & Status of Each

At the moment there are 4 acitvely developed and maintained "ports" of PySimpleGUI. These include:

1. tkinter - Fully complete
2. Qt using Pyside2 - Alpha stage. Not all features for all Elements are done
3. WxPython - Development stage, pre-releaser. Not all Elements are done. Some known problems with multiple windows
4. Remi (Web browser support) - Development stage, pre-release.

While PySimpleGUI, the tkinter port, is the only 100% completed version of PySimpleGUI, the other 3 ports have a LOT of functionality in them and are in active use by a large portion of the installations. You can see the number of Pip installs at the very top of this document to get a comparison as to the size of the install base for each port. The "badges" are right after the logo.

The PySimpleGUI "Family"

What's The Big Deal? What is it?

PySimpleGUI wraps tkinter, Qt, WxPython and Remi so that you get all the same widgets, but you interact with them in a more friendly way that's common across the ports.

What does a wrapper do (Yo! PSG in the house!)? It does the layout, boilerplate code, creates and manages the GUI Widgets for you and presents you with a **simple, efficient interface.** Most importantly, it maps the Widgets in tkinter/Qt/Wx/Remi into PySimpleGUI Elements. Finally, it replaces the GUIs' event loop with one of our own.

You've seen examples of the code already. The big deal of all this is that anyone can create a GUI simply and quickly that matches GUIs written in the native GUI framework. You can create complex layouts with complex element interactions. And, that code you wrote to run on tkinter will also run on Qt by changing your import statement.

If you want a deeper explanation about the architecture of PySimpleGUI, you'll find it on ReadTheDocs in the same document as the Readme & Cookbook. There is a tab at the top with labels for each document.

The "Ports"

There are distinct ports happening as mentioned above. Each have their own location on GitHub under the main project. They have their own Readme with is an *augmentation* of this document... they are meant to be used together.

PySimpleGUI is released on PyPI as 5 distinct packages. 1. PySimpleGUI - tkinter version 2. PySimpleGUI27 - tkinter version that runs on 2.7 3. PySimpleGUIWx - WxPython version 4. PySimpleGUIQt - PySided2 version 5. PySimpleGUIWeb - The web (Remi) version

You will need to install them separately

There is also an accompanying debugger known as `imwatchingyou`. If you are running the tkinter version of PySimpleGUI, you will not need to install the debugger as there is a version embedded directly into PySimpleGUI.

Qt Version

Qt was the second port after tkinter. It is the 2nd most complete with the original PySimpleGUI (tkinter) being the most complete and is likely to continue to be the front-runner. All of the Elements are available on PySimpleGUIQt.

As mentioned previously each port has an area. For Qt, you can learn more on the PySimpleGUIQt GitHub site. **There is a separate Readme file for the Qt version** that you'll find there. This is true for all of the PySimpleGUI ports.

Give it a shot if you're looking for something a bit more "modern". PySimpleGUIQt is currently in **Alpha**. *All of the widgets are operational but some may not yet be full-featured.* If one is missing and your project needs it, log an Issue. It's how new features are born.

Here is a summary of the Qt Elements with no real effort spent on design clearly. It's an example of the "test harness" that is a part of each port. If you run the PySimpleGUI.py file itself then you'll see one of these tests.

As you can see, you've got a full array of GUI Elements to work with. All the standard ones are there in a single window. So don't be fooled into thinking PySimpleGUIQt is barely working or doesn't have many widgets to choose from. You even get TWO "Bonus Elements" - `Dial` and `Stretch`

WxPython Version

PySimpleGUIWx GitHub site. **There is a separate Readme file for the WxPython version**.

Started in late December 2018 PySimpleGUIWx started with the SystemTray Icon feature. This enabled the package to have one fully functioning feature that can be used along with tkinter to provide a complete program. The System Tray feature is complete and working very well. It was used not long ago in a corporate setting and has been performing with few problems reported.

The Windowing code was coming together with Reads operational. The elements were getting completed on a regular basis. But I ran into multiwindow problems. And it was at about this time that Remi was suggested as a port.

Remi (the "web port") overnight lept the WxPython effort and Web became a #1 priority and continues to be. The thought is that the desktop was well represented with PySimpleGUI, PySimpleGUIQt, and PySimpleGUIWx. Between those ports is a solid winowing system and 2 system tray implementations and a nearly feature complete Qt effort. So, the team was switched over to PySimpleGUIWeb.

Web Version (Remi)

PySimpleGUIWeb GitHub site. **There is a separate Readme file for the Web version**.

New for 2019, PySimpleGUIWeb. This is an exciting development! PySimpleGUI in your Web Browser!

The underlying framework supplying the web capability is the Python package Remi. https://github.com/dddomodossola/remi Remi provides the widgets as well as a web server for you to connect to. It's an exiting new platform to be running on and has temporarily bumped the WxPython port from the highest priority. PySimpleGUIWeb is the current high priority project.

Use this solution for your Pi projects that don't have anything connected in terms of input devices or display. Run your Pi in "headless" mode and then access it via the Web interface. This allows you to easily access and make changes to your Pi without having to hook up anything to it.

It's not meant to "serve up web pages"

PySimpleGUIWeb is first and foremost a **GUI**, a program's front-end. It is designed to have a single user connect and interact with the **GUI**.

If more than 1 person connects at a time, then both users will see the exact same stuff and will be interacting with the program as if a single user was using it.

Source code compatibility

In theory, your source code is completely portable from one platform to another by simply changing the import statement. That's the GOAL and suprisingly many times this 1-line change works. Seeing your code run on tkinter, then change the import to `import PySimpleGUIWeb as sg` and instead of a tkinter window, up pops your default browser with your window running on it is an incredible feeling.

But, ***caution is advised.*** As you've read already, some ports are further along than others. That means when you move from one port to another, some features may not work. There also may be some alignment tweaks if you have an application that precisely aligns Elements.

What does this mean, assuming it works? It means it takes a trivial amount of effort to move across GUI Frameworks. Don't like the way your GUI looks on tkinter? No problem, change over to try PySimpleGUIQt. Made a nice desktop app but want to bring it to the web too? Again, no problem, use PySimpleGUIWeb.

repl.it Version

Want to really get your mind blown? Check out this PySimpleGUI program running in your web browser.

Thanks to the magic of repl.it and Remi it's possible to run PySimpleGUI code in a browser window without having Python running on your computer. This should be viewed as a teaching and demonstration aid. It is not meant to be a way of serving up web pages. It wouldn't work

any way as each user forks and gets their own, completely different, workspace.

There are 2 ports of PySimpleGUI that run on repl.it - PySimpleGUI and PySimpleGUIWeb.

PySimpleGUI (tkinter based)

The primary PySimpleGUI port works very well on repl.it due to the fact they've done an outstanding job getting tkinter to run on these virtual machines. Creating a program from scratch, you will want to choose the "Python with tkinter" project type.

The virtual screen size for the rendered windows isn't very large, so be mindful of your window's size or else you may end up with buttons you can't get to.

You may have to "install" the PySimpleGUI package for your project. If it doesn't automatically install it for you, then click on the cube along the left edge of the browser window and then type in PySimpleGUI or PySimpleGUIWeb depending on which you're using.

PySimpleGUIWeb (Remi based)

For PySimpleGUIWeb programs you run using repl.it will automatically download and install the latest PySimpleGUIWeb from PyPI onto a virtual Python environment. All that is required is to type `import PySimpleGUIWeb` you'll have a Python environment up and running with the latest PyPI release of PySimpleGUIWeb.

Creating a repl.it project from scratch / troubleshooting

To create your own repl.it PySimpleGUI project from scratch, first choose the type of Python virtual machine you want. For PySimpleGUI programs, choose the "Python with tkinter" project type. For PySimpleGUIWeb, choose the normal Python project.

There have been times where repl.it didn't do the auto import thing. If that doesn't work for some reason, you can install packages by clicking on the package button on the left side of the interface, typing in the package name (PySimpleGUI or PySimpleGUIWeb) and install it.

Why this is so cool (listen up Teachers, tutorial writers)

Educators in particular should be interested. Students can not only post their homework easily for their teacher to access, but teachers can also run the students programs online. No downloading needed. Run it and check the results.

For people wanting to share their code, especially when helping someone with a problem, it's a great place to do it. Those wishing to see your work do not have to be running Python nor have PySimpleGUI installed.

The way I use it is to first write my PySimpleGUI code on Windows, then copy and paste it into Repl.it.

Finally, you can embed these Repl.it windows into web pages, forum posts, etc. The "Share" button is capable of giving you the block of code for an "iframe" that will render into a working repl.it program in your page. It's amazing to see, but it can be slow to load.

Repl.it is NOT a web server for you to "deploy" applications!

Repl.it is not meant to serve up applications and web pages. Trying to use it that way will not ressult in satisfactory results. It's simply too slow and too technical of an interface for trying to "deploy" using it. PySimpleGUIWeb isn't a great choice in serving web pages. It's purpose is more to build a GUI that runs in a browser.

Macs

It's surprising that Python GUI code is completely cross platform from Windows to Mac to Linux. No source code changes. This is true for both PySimpleGUI and PySimpleGUIQt.

However, **Macs** suck. They suck on tkinter in particular. The "Look and feel" calls are disabled for Macs. Colored buttons in particular are broken. And, you are unable to specify filetypes when using the FileBrowse button. None of this is PySimpleGUI code issues, of course, they're all in tkinter. Consider using Qt instead of tkinter on the Mac. Or, if using tkinter, create your own button images.

Look through the open and closed issues if you're a Mac person with a problem. It's highly likely the problem has been at least seen previously and there may even be a fix or workaround

Switching to "Light Mode" is known to fix some of the problems. They honestly are tkinter/Mac problems.

Make sure you're running tkinter 8.6 and up. However, as I'm learning 8.6 can mean almost anything as the minor release numbers are not provided (8.6.1, etc). Turns out 8.6 has been in development for YEARS. The 8.6 that comes with Python 3.7.4 doesn't support table colors for example even though it's version 8.6.

Support

Don't Suffer Silently

The GitHub Issues are checked *often*. Very often. **Please** post your questions and problems there and there only. Please don't post on Reddit, Stackoverflow, on forums, until you've tried posting on the GitHub.

Why? *It will get you the best support possible.* Second, you'll be helping the project as what you're experiencing might very well be a bug, or even a *known* bug. Why spend hours thrashing, fighting against a known bug?

It's not a super-buggy package, but users do experience problems just the same. Maybe something's not explained well enough in the docs. Maybe you're making a common mistake. Maybe that feature isn't complete yet.

You won't look stupid posting an Issue on GitHub. It's just the opposite.

How to log issues

PySimpleGUI is an active project. Bugs are fixed, features are added, often. Should you run into trouble, **open an issue** on the GitHub site and you'll receive help. Posting questions on StackOverflow, Forums, Mailing lists, Reddit, etc, is not the fastest path to support and taking it may very well lead you astray as folks not familiar with the package struggle to help you. You may also run into the common response of "I don't know PySimpleGUI (and perhaps dislike it as a result), but I know you can do that with Qt".

Why only 1 location? It's simple.... it's where the bugs, enhancements, etc are tracked. It's THE spot on the Internet for this project. It's not some freakish being in control, telling people how to do things. It's so that YOU get the best and quickest support possible.

So, open an Issue, choose "custom form" and fill it out completely. There are very good reasons behind all of the questions. Cutting corners only cuts your chances of getting help and getting quality help as it's difficult enough to debug remotely. Don't handicap people that want to help by not providing enough information.

Be sure and run your program outside of your IDE *first*. Start your program from the shell using `python` or `python3` command. On numerous occassions much time was spent chasing problems caused by the IDE. By running from a command line, you take that whole question out of the problem, an important step.

Don't sit and stew, trying the same thing over and over, until you hate life... stop, and post an Issue on the GitHub. Someone **WILL** answer you. Support is included in the purchase price for this package (the quality level matches the price as well I'm afraid). Just don't be too upset when your free support turns out to be a little bit crappy, but it's free and typically good advice.

PySimpleGUI Trolls

Yea, they're out there. Would have NEVER in a billion years guessed that there would be objection to this package, coming from a small, but vocal, number of people. I naively believed everyone would be supportive, see the value, hold hands around the fire, sing songs, and in the very least be truthful. But, this ain't Kansas and the Internet is well...

If someone is actively discouraging you from using this package, then know you are on the right track and you should give it a try. Stuff like this sometimes happens because the person works for a particular company or they have a GUI package of their own.... that's not popuplar, or they like to tear things down or I dunno... people can be really weird and emotional.

I promise you're not going to be wrecked for life. It will not cause you to be a bad programmer with bad habits. It will not ruin your career. It's not going to teach you bad habits. One person I know got a bonus based on a PySimpleGUI program he wrote.

How about success being the goal? Spend time polishing your GUI and working on your primary code instead of struggling your way through the mountains of documentation in the Qt library, trying to set the color of something. How about getting your program done, working, and in use?

Start with PySimpleGUI, then in the future if you want to code directly in Qt, it's not like you won't be able to learn something else due to your PySimpleGUI knowledge. Unable to write a `while` loop because you already learned using a `for` loop? That seems to be the logic.

If anything, you'll have more knowledge than most people that are just starting because you'll have already built working GUIs, perhaps many of them and understand how to layout an efficient interface as well as having a good education in GUI Widgets and how they work.

Here are the forces likely at work.... said Troll has been programming for a while now and really knows these GUI frameworks. Probably invested a lot of hours learning them and thus has some marketable skills (yes, follow the money).

Enter snotty-nosed high-shool or first year programmer using PySimpleGUI that manages to make GUI windows with great ease... that look, well, pretty much the **same** as the windows Sir Troll ***was*** used to getting paid big bucks to make. What used to be a skill only a select few can do, now 1,000's, 10,000's, or 100,000s more programmers can do, quicker and easier. Bummer Mr. Troll. Bummer.

"***It's supposed to be difficult to write a GUI***" was an actual reason listed why not to use PySimpleGUI by one person offering to help junior programmers learn Qt. Really? It's supposed to be difficult. Huh. Well, not in this Python universe. Maybe in C++ world it works that way??

Just know you're safe in trying and possibly even succeeding in the process.

Target Audience

PySimpleGUI is trying to serve the 80% of GUI *problems*. The other 20% go straight to tkinter, Qt, WxPython, Remi, or whatever fills that need. That 80% is **a huge problem space**.

The "Simple" of PySimpleGUI describes how easy it is to use, not the nature of the problem space it solves. Note that people are not part of that description. It's not trying to solve GUI problems for 80% of the people trying it. PySimpleGUI tries to solve 80% of GUI ***problems***, regardless of the programmer's experience level.

Is file I/O in Python limited to only certain people? Is starting a thread, building a multi-threaded Python program incredibly difficult such that it takes a year to learn? No. It's quite easy. Like most things Python, you import the object from package and you use it. It is 2 lines of Python code to create and start a thread.

Why can't it be 2 lines of code to show a GUI window? What's SO special about the Python GUI libraries that they require you to follow a specific Object Oriented model of development? Other parts and packages of Python don't tend to do that.

The reason is because they didn't originate in Python. They are strangers in a strange land and they had to be "adapted". They started as C++ programs / SDKs, and remain that way too. There's a vaneer of Python slapped onto the top of them, but that sure didn't make them fit the language as well as they could have.

PySimpleGUI is designed with both the beginner and the experienceed developer in mind. Why? Because both tend to like compact code. Most like people, we just want to get sh*t done, right? And, why not do it in a way that's like how most of Python works?

The beginners can begin working with GUIs ***in their first week of Python education***. The professionals can jump right into the deep end of the pool to use the entire array of Elements and their capabilities to build stuff like a database application.

Here's a good example of how PySimpleGUI serves these 2 groups.... the `InputText` Element has 16 potential parameters, yet you'll find 0 or 1 parameters set by beginners. Look at the examples throughout this document and you'll see the code fragments utilize a tiny fraction of the potential parameters / settings. Simple... **keep it simple for the default case**. This is part of the PySimpleGUI mission.

Some developers are heavily wedded to the existing GUI Framework Architectures (Qt, WxPyton, tkinter). They like the existing GUI architectures (they're all roughly the same, except this one). If you're in that crowd, join the "20% Club" just down the street. There's plenty of room there with plenty of possible solutions.

But how about a quick stop-in for some open mindedness exercises. Maybe you will come up with an interesting suggestion even if you don't use it. Or maybe PySimpleGUI does something that inspires you to write something similar directly in Qt. And please, at least be civil about it.

There is room for multiple architectures. Remember, you will not be *harmed* by writing some PySimpleGUI code just like you won't by writing some tkinter or Qt code. Your chances of feeling harmed is more likely from one of those 2.

Beginners & Easier Programs

There are a couple of reasons beginners stop in for a look. The first is to simply throw a simple GUI onto the front of an existing command line application. Or maybe you need to popup a box to get a filename. These can often be simple 1-line `Popup` calls. Of course, you don't have to be a beginner to add a GUI onto one of your existing command line programs. Don't feel like because you're an advanced programmer, you need to have an advanced solution.

If you have a more intricate, complete, perhaps multi-window design in mind, then PySimpleGUI still could be your best choice.

This package is not only great to use as your first GUI package, but it also teaches how to design and utilize a GUI. It does it better than the existing GUIs by removing the syntax, and lengthy code that can take an otherwise very simple appearing program into something that's completely unrecognizable. With PySimpleGUI your 'layout' is all you need to examine to see the different GUI Elements that are being used.

Why does PySimpleGUI make it any easier to learn about GUIs? Because it removes the classes, callback functions, object oriented design to better get out of your way and let you focus entirely on your GUI and not how to represent it in code.

The result is 1/2 to 1/10 th the amount of code that implements the exact same layout and widgets as you would get from coding yourself directly in Qt5. It's been tested many times... again and again, PySimpleGUI produces significantly less code than Qt and the frameworks it runs on.

Forget syntax completely and just look on the overall activities of a PySimpleGUI programmer. You have to design your window.... determine your inputs and your outputs, place buttons in strategic places, create menus, ... You'll be busy just doing all those things to design and define your GUI completely independent upon the underlying framework.

After you get all those design things done and are ready to build your GUI, it's then that you face the task of learning a GUI SDK. Why not start with the easy one that gives you many successes? You're JUST getting ***started***, so cut yourself a break and use PySimpleGUI so that you can quickly get the job done and move on to the next GUI challenge.

Advanced Programmers, Sharp Old-Timers, Code Slingers and Code Jockeys

It's not perfect, but PySimpleGUI is an amazing bit of technology. It's the programmer, the computer scientist, that has experience working with GUIs in the past that will recognize the power of this simple architecture.

What I hear from seasoned professionals is that PySimpleGUI saves them a **ton** of time. They've written GUI code before. They know how to lay out a window. These folks just want to get their window working and quick.

With the help of IDE's like PyCharm, Visual Studio and Wing (the offically supported IDE list) you get instant documentation on the calls you are making. On PyCharm you instantly see both the call signature but also the explanations about each parameter.

If the screenshots, demo programs and documentation don't convince you to at least **give it a try, once**, then you're way too busy, or ..... I dunno, I stopped guessing "why?" some time ago.

Some of the most reluctant of people to try PySimpleGUI have turned out to be some of the biggest supporters.

A Moment of Thanks To The PySimpleGUI Users

I want to thank the early users of PySimpleGUI that started in 2018. Your suggestions helped shape the package and have kept it moving forward at a fast pace.

For all the users, while I can't tell you the count of the number of times someone has said "thank you for PySimpleGUI" as part of logging and Issue, or a private message or email, but I can tell you that it's been significant.

EVERY one of those "thank you" phrases, no matter how small you may think it is, helps tremendously.

Sometimes it's what gets me past a problem or gets me to write yet more documentation to try and help people understand quicker and better. Let's just say the effect is always positive and often significant.

PySimpleGUI users have been super-nice. I doubt all Open Source Projects are this way, but I could be wrong and every GitHub repository has awesome users. If so, that's even more awesome!

THANK YOU PySimpleGUI USERS!

Learning Resources

This document.... you must be willing to read this document if you expect to learn and use PySimpleGUI.

If you're unwilling to even try to figure out how to do something or find a solution to a problem and have determined it's "easier to post a question first than to look at the docs", then this is not the GUI package for you. *If you're unwilling to help yourself, then don't expect someone else to*

try first. You need to hold up your end of the bargain by at least doing some searches of this document.

While PySimpleGUI enables you to write code easily, it doesn't mean that it magically fills your head with knowledge on how to use it. The built-in docstrings help, but they can only go so far.

Searching this document is as easy as pressing Control + F.

This document is on the GitHub homepage, as the readme. http://www.PySimpleGUI.com will get you there. If you prefer a version with a Table of Contents on the left edge then you want to go to http://www.PySimpleGUI.org .

The PySimpleGUI, Developer-Centric Model

You may think that you're being fed a line about all these claims that PySimpleGUI is built specifically to make your life easier and a lot more fun than the alternatives.... especially after reading the bit above about reading this manual.

Psychological Warfare

Brainwashed. Know that there is an active campaign to get you to be successful using PySimpleGUI. The "Hook" to draw you in and keep you working on your program until you're satisfied is to work on the dopamine in your brain. Yes, your a PySimpleGUI rat, pressing on that bar that drops a food pellet reward in the form of a working program.

The way this works is to give you success after success, with very short intervals between. For this to work, what you're doing must work. The code you run must work. Make small changes to your program and run it over and over and over instead of trying to do one big massive set of changes. Turn one knob at a time and you'll be fine.

Find the keyboard shortcut for your IDE to run the currently shown program so that running the code requires 1 keystroke. On PyCharm, the key to run what you see is Control + Shift + F10. That's a lot to hold down at once. I programmed a hotkey on my keyboard so that it emits that combination of keys when I press it. Result is a single button to run.

Tools

These tools were created to help you achieve a steady stream of these little successses.

- This readme and its example pieces of code
- The Cookbook - Copy, paste, run, success
- Demo Programs - Copy these small programs to give yourself an instant headstart
- Documentation shown in your IDE (docstrings) means you do not need to open any document to get the full assortment of options available to you for each Element & function call

The initial "get up and running" portion of PySimpleGUI should take you less than 5 minutes. The goal is 5 minutes from your decision "I'll give it a try" to having your first window up on the screen "Oh wow, it was that easy?!"

The primary learning pathes for PySimpleGUI are:

- This readme document over 100 pages of PySimpleGUI User Manual
  http://www.PySimpleGUI.org
- The Cookbook - Recipes to get you going and quick
  http://Cookbook.PySimpleGUI.org
- The Demo Programs - Start hacking on one of these running soluitions
  http://www.PySimpleGUI.com

Everything is geared towards giving you a "quick start" whether that be a Recipe or a Demo Program. The idea is to give you something running and let you hack away at it. As a developer this saves tremendous amounts of time.

You **start** with a working program, a GUI on the screen. Then have at it. If you break something (`"a happy little accident"` as Bob Ross put it), then you can always backtrack a little to a known working point.

A high percentage of users report both learning PySimpleGUI and completing their project in a single day.

This isn't a rare event and it's not bragging. GUI programming doesn't HAVE to be difficult by definition and PySimpleGUI has certainly made it much much more approachable and easier (not to mention simpler).

But, you need to look at this document when pushing into new, unknown territory. Don't guess... or more specifically, don't guess and then give up when it doesn't work.

This Readme and Cookbook

The readme and Cookbook, etc are best viewed on ReadTheDocs. The quickest way there is to visit: http://www.PySimpleGUI.org

You will be auto-forwarded to the right destination. There are multiple tabs on ReadTheDocs. One for the main readme and one for the Cookbook. There are other documents there like an architectural design doc.

The Cookbook has approx 27 "Recipes" or short programs that can be easily copied and pasted.

Demo Programs

The GitHub repo has the Demo Programs. There are ones built for plain PySimpleGUI that are usually portable to other versions of PySimpleGUI. And there are some that are associated with one of the other ports. The easiest way to the GitHub:

http://www.PySimpleGUI.com

As of this writing, on 2019-07-10 there are 177 Demo Programs for you to choose from.

These programs demonstrate to you how to use the Elements and especially how to integtate PySimpleGUI with some of the popular open source technologies such as OpenCV, PyGame, PyPlot, and Matplotlib to name a few.

Many Demo Programs that are in the main folder will run on multiple ports of PySimpleGUI. There are alse port-specific Demo Programs. You'll find those in the folder with the port. So, Qt specific Demo Programs are in the PySimpleGUIQt folder.

The Quick Tour

Let's take a super-brief tour around PySimpleGUI before digging into the details. There are 2 levels of windowing support in PySimpleGUI - High Level and Customized.

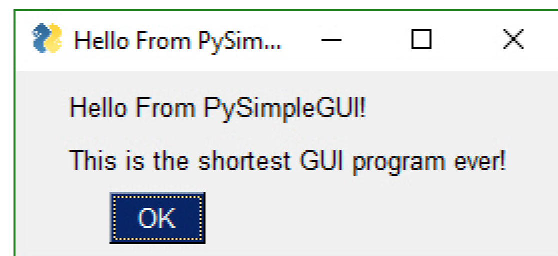
The high-level calls are those that perform a lot of work for you. These are not custom made windows (those are the other way of interacting with PySimpleGUI).

Let's use one of these high level calls, the `Popup` and use it to create our first window, the obligatory "Hello World". It's a single line of code. You can use these calls like print statements, adding as many parameters and types as you desire.

```
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? No kidding this is a valid program and it uses Elements and produce the same Widgets like you normally would in a tkinter program. It's just been compacted together is all, strictly for demonstration purposes as there's no need to go that extreme in compactness, unless you have a reason to and then you can be thankful it's possible to do.




```
import PySimpleGUI as sg

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

The Beauty of Simplicity

> One day I will find the right words, and they will be simple. — Jack Kerouac

That's nice that you can crunch things into 1 line, like in the above example, but it's not readable. Let's add some whitespace so you can see the **beauty** of the PySimpleGUI code.

Take a moment and look at the code below. Can you "see" the window looking at the `layout` variable, knowing that each line of code represents a single row of Elements? There are 3 "rows" of Elements shown in the window and there are 3 lines of code that define it.

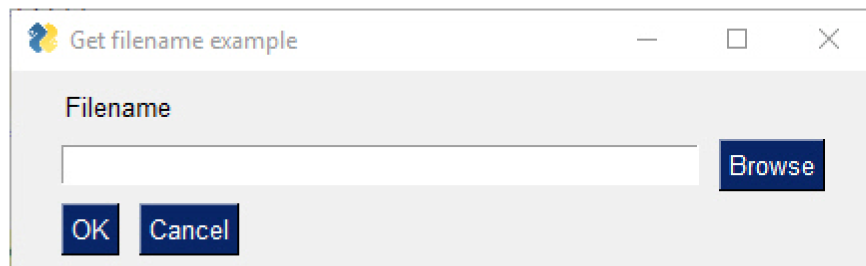
Creating and reading the user's inputs for the window occupy the last 2 lines of code, one to create the window, the last line shows the window to the user and gets the input values (what button they clicked, what was input in the Input Element)

```
import PySimpleGUI as sg

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

window = sg.Window('Get filename example', layout)

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



Unlike other GUI SDKs, you can likely understand every line of code you just read, even though you have not yet read a single instructional line from this document about how you write Elements in a layout.

There are no pesky classes you are *required* to write, no callback functions to worry about. None of that is required to show a window with some text, an input area and 2 buttons using PySimpleGUI.

The same code, in tktinter, is 5 times longer and I'm guessing you won't be able to just read it and understand it. While you were reading through the code, did you notice there are no comments, yet you still were able to understand, using intuition alone.

You will find this theme of Simple everywhere in and around PySimpleGUI. It's a way of thinking as well as an architecture direction. Remember, you, Mr./Ms. Developer, are at the center of the package. So, from your vantage point, of course everything should look and feel simple.

Not only that, it's the Pythonic thing to do. Have a look at line 3 of the "Zen of Python".

> The Zen of Python, by Tim Peters
>
> Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one-- and preferably only one -- obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than *right* now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea -- let's do more of those!

I just hope reading all these pages of documentation is going to make you believe that we're breaking suggestion:

> If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea.

I don't think PySimpleGUI is ***difficult*** to explain, but I am striving to fully explain it so that you don't do this:

> In the face of ambiguity, refuse the temptation to guess.

Sometimes you can guess and be fine. Other times, things may work, but the side effects are potentially significant. There may be a much better way to solve a problem - Log an Issue on GitHub!

Some Examples

Polishing Your Windows = Building "Beautiful Windows"

And STILL the Zen of Python fits:

> Beautiful is better than ugly.

but this fits too:

> Although practicality beats purity.

Find a balance that works for you.

"But tkinter sucks" "It looks like the 1990s" (this one is often said by people that were not alive in the 1990s) "What Python GUI SDK will make my window look beautiful?" (posted to Reddit at least every 2 weeks)

These windows below were ALL made using PySimpleGUI, the tkinter version and they look good enough to not be simply scoffed at and dismissed. Remember, developer, you have a rather significant hand in how your application looks and operates. You certainly cannot pin it all on the GUIs you're using.

So many posts on Reddit asking which GUI is going to result in a "beautiful window", as if there's a magic GUI library that pretties things up for you. There are some calls in PySimpleGUI that will help you. For example, you can make a single call to "Chang the look and feel" which loads predefined color pallets so your windows can have some instant color and it matches.

Beautiful windows are created, not simply given to you. There are people that design and create artwork for user interfaces, you know that right? Artists draw buttons, artwork that you include in the window to make it nicer.

Some of these have been "polished", others like the Matplotlib example is more a functional example to show you it works.

Batter Up!

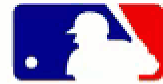
Pitcher: Robert Frost	ODDS	
Hit	20-1	
Out	2-1	✔
Walk	1-2	

Batter: Truman Capote	ODDS	
Hit	20-1	✔
Out	2-1	
Walk	1-2	

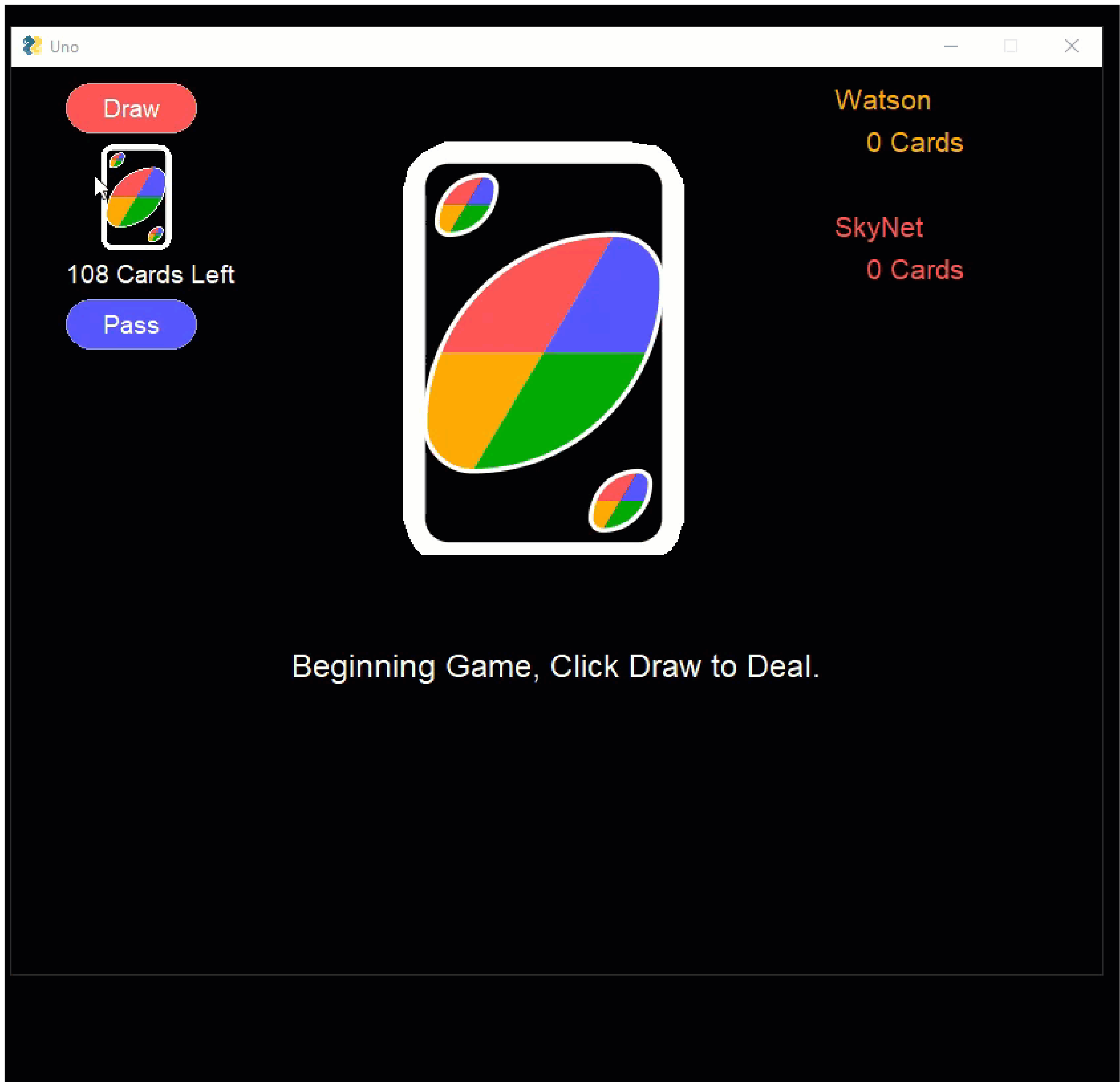
Credits Available

0

PLACE YOUR BET



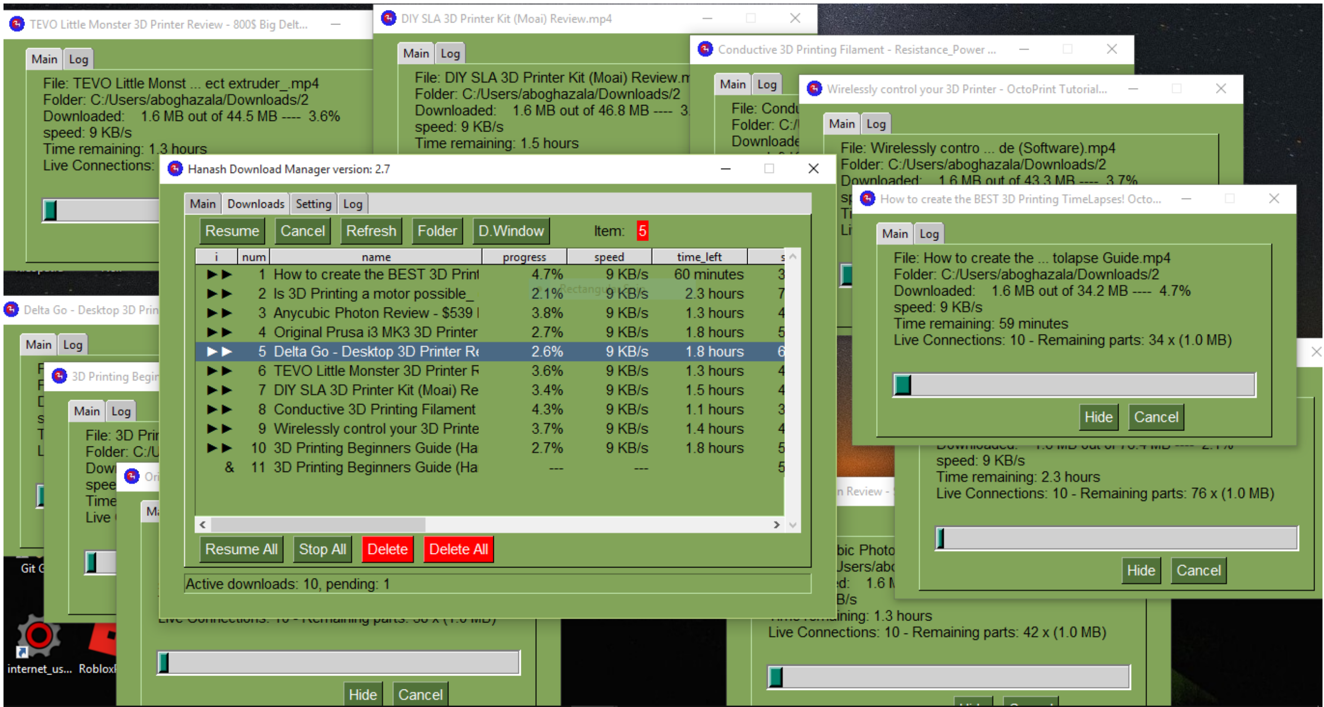
WIN! BET LOSE!

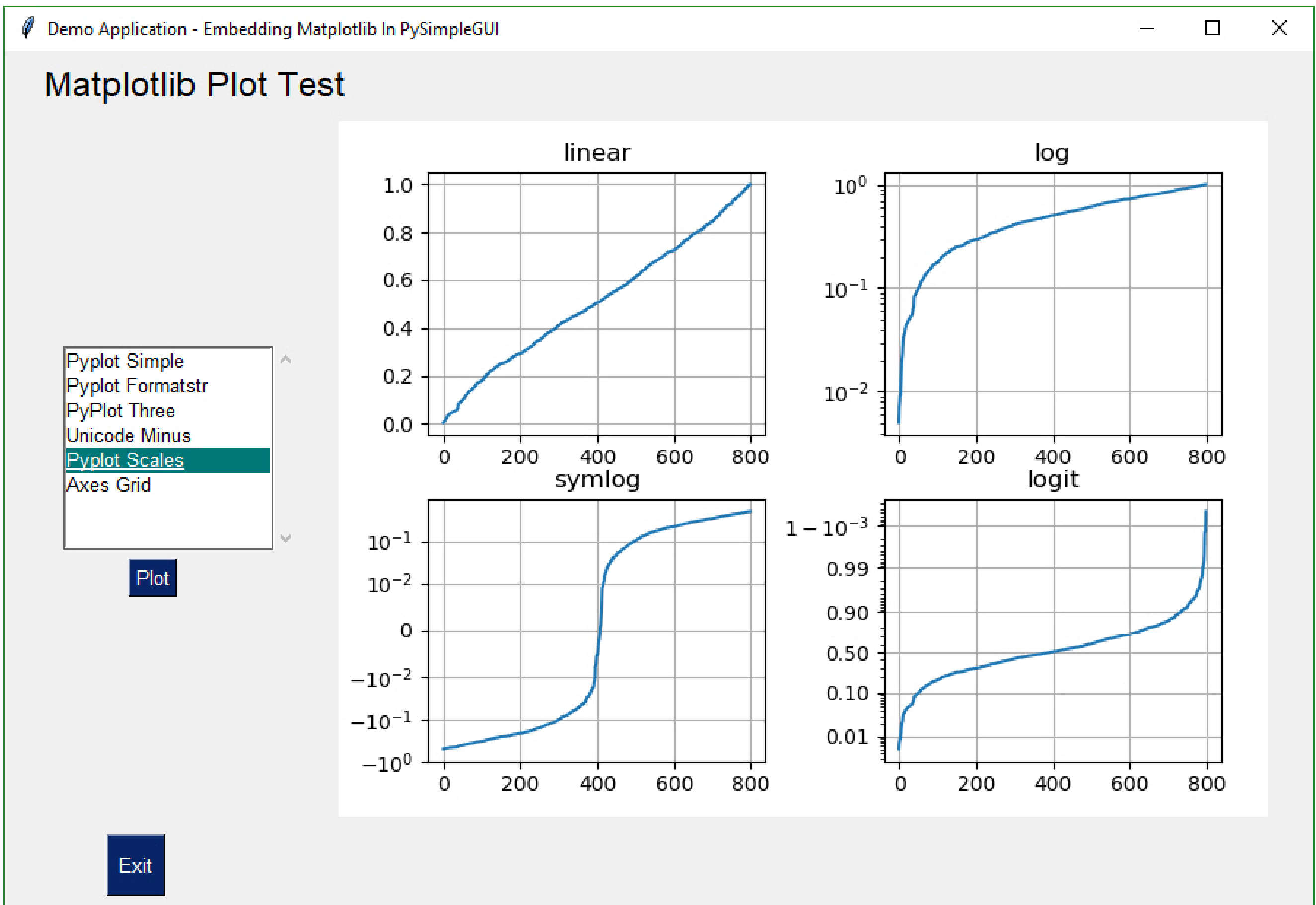


This chess program is capable of running multiple AI chess engines and was written by another user using PySimpleGUI.

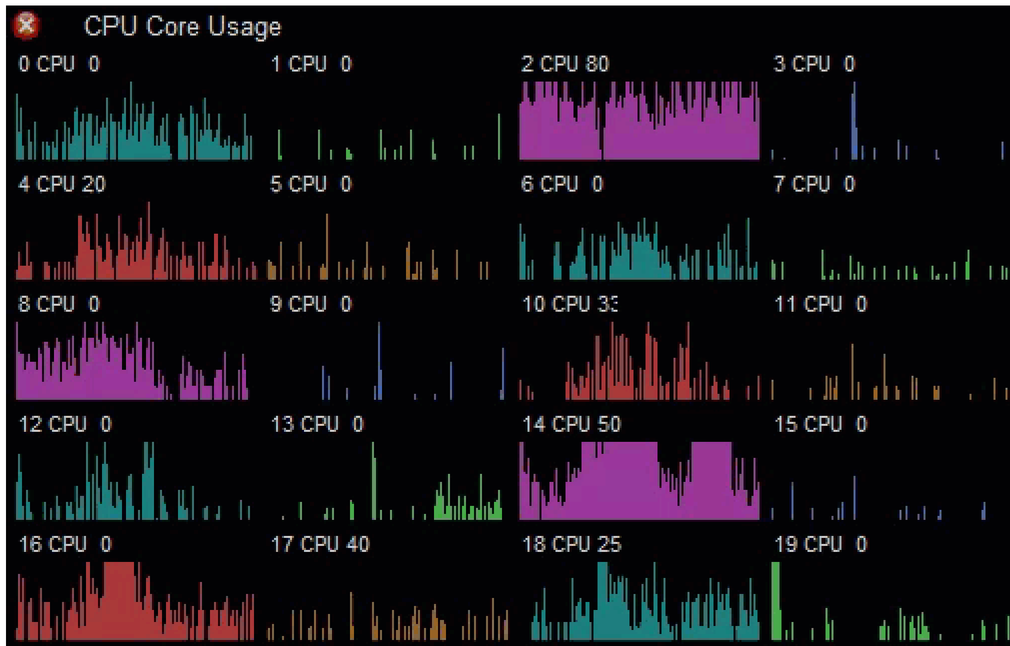
This downloader can download files as well as YouTube videos and metadata. If you're worried about multiple windows working, don't. Worried your project is "too much" or "too complex" for PySimpleGUI? Do an initial assessment if you want. Check out what others have done.

Your program have 2 or 3 windows and you're concerned? Below you'll see 11 windows open, each running independently with multiple tabs per window and progress meters that are all being updated concurrently.





Make beautiful looking, alpha-blended (partially transparent) Rainmeter-style Desktop Widgets that run in the background.



Want to build a Crossword Puzzle? No problem, the drawing primitives are there for you.

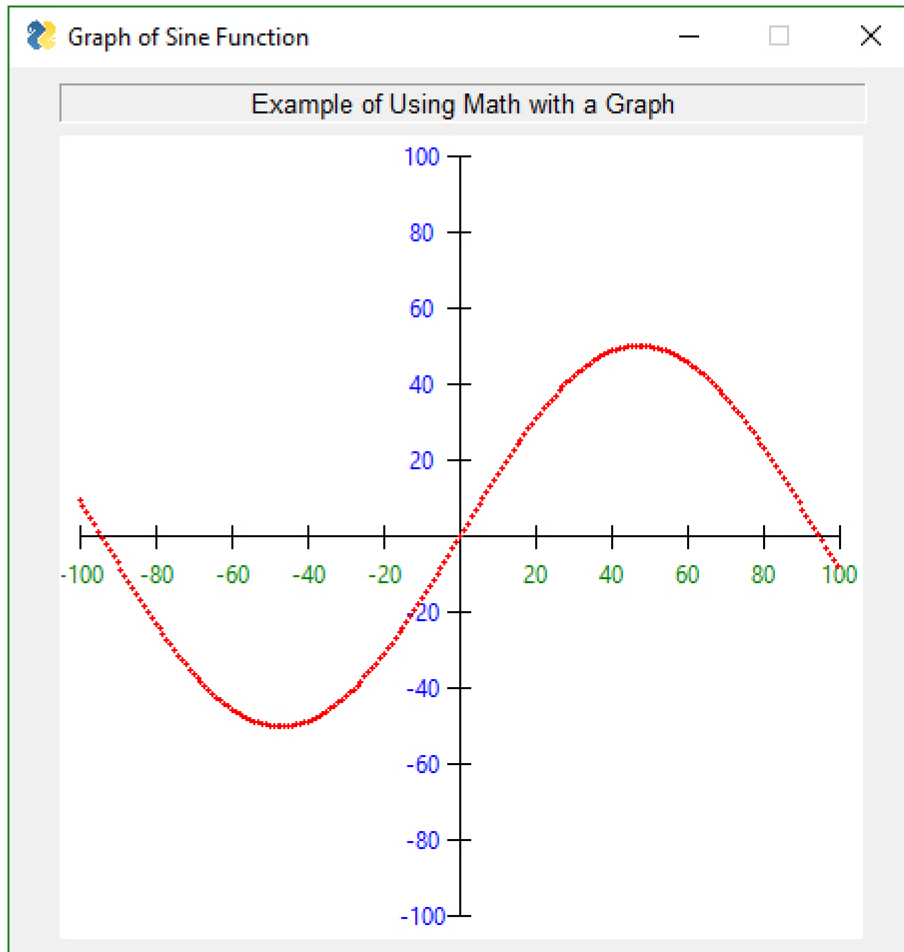
Window Title

Crossword Puzzle Using PySimpleGUI

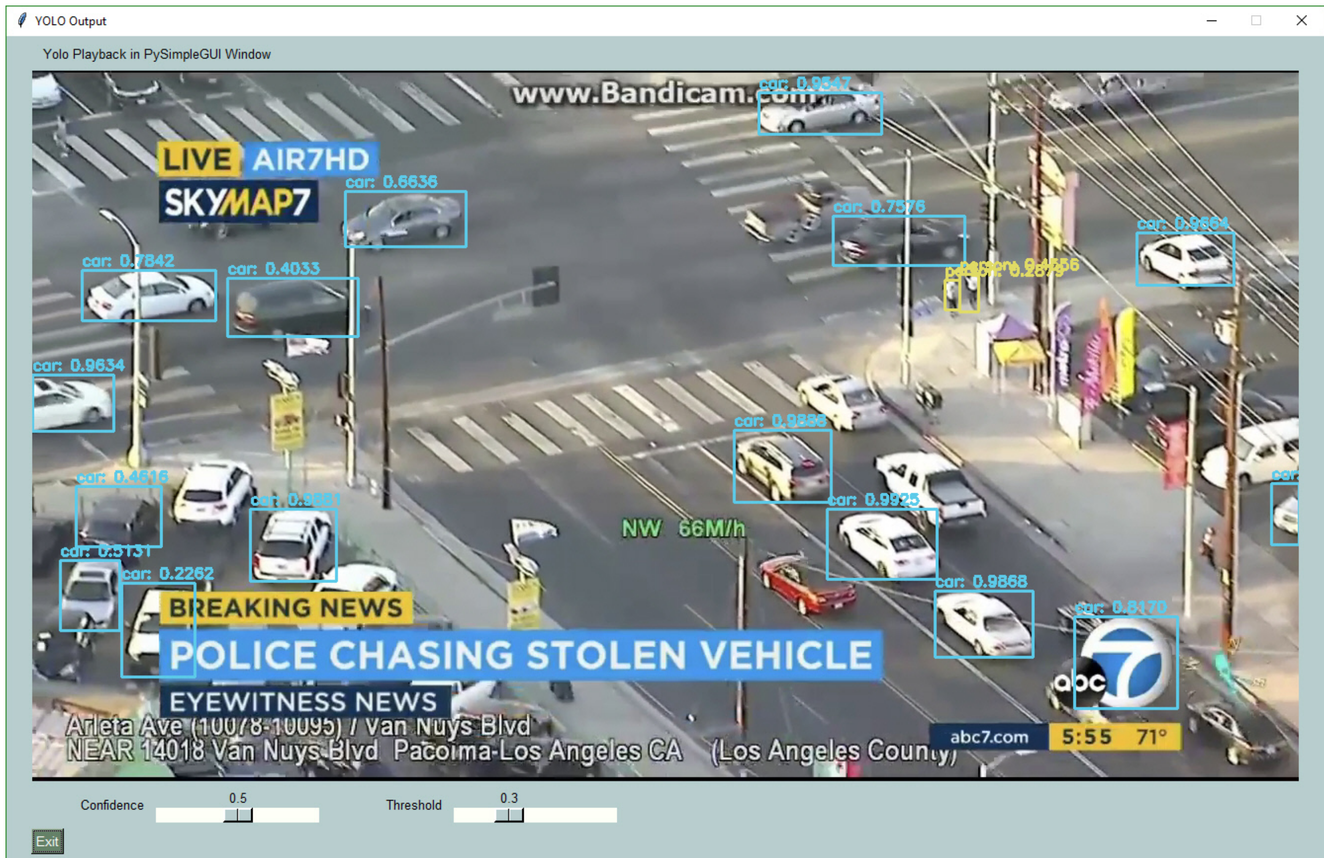
1 L	2 W	3 L	4 P	5 U	6 V	7 S	8 N	9 J	10 O		12	13	14	15	16
7	8	9 U	10 J		12	13 X	14	15	16	17	18	19	20	21	22
13	14	15		17	18	19 I	20		22	23	24	25	26	27	28
19	20	21	22	23	24	25 L	26	27	28	29	30	31	32	33	34
25	26	27	28	29	30	31 E	32	33	34			37	38	39	
31	32	33 R	34	35	36	37 D	38	39	40	41	42	43		45	46
37	38	39 Z	40	41		43 B		45	46	47	48		50	51	52
43	44	45 A	46	47	48	49 S	50	51	52	53		55	56	57	58
49	50	51 H	52	53		55 D		57	58	59	60	61	62	63	64
55	56	57	58	59	60	61 L	62	63	64	65	66	67	68	69	70
61	62	63	64	65	66		68	69	70	71	72	73		75	76
67	68	69	70	71	72	73	74	75		77	78	79	80	81	
73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88
		81	82	83	84	85	86	87	88	89		91		93	
85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106

Show Exit

There are built-in drawing primitives

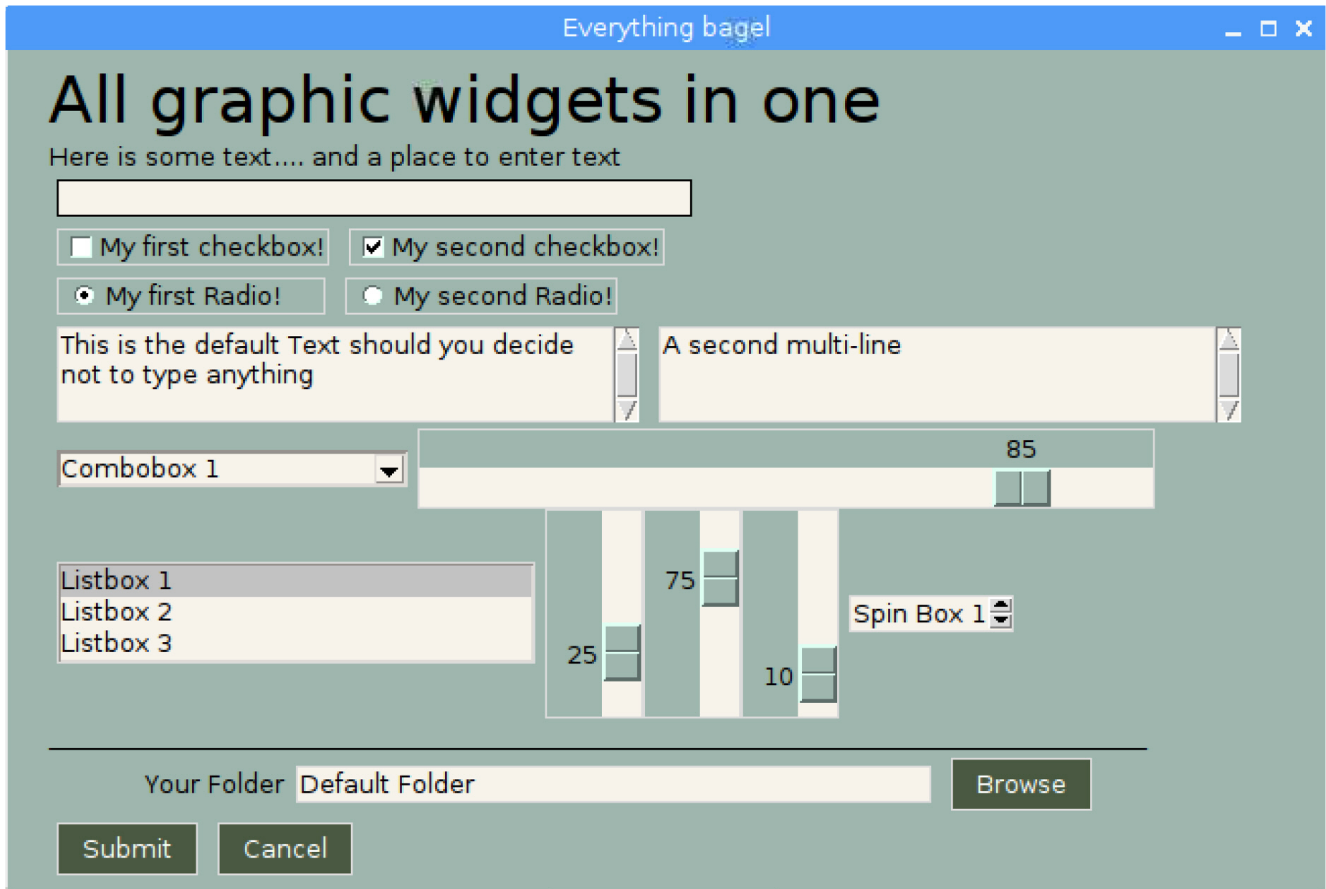


Frame from integration with a YOLO Machine Learning program that does object identification in realtime while allowing the user to adjust the algorithms settings using the sliders under the image. This level of interactivity with an AI algorithm is still unusual to find due to difficulty of merging the technologies of AI and GUI. It's no longer difficult. This program is under 200 lines of code.

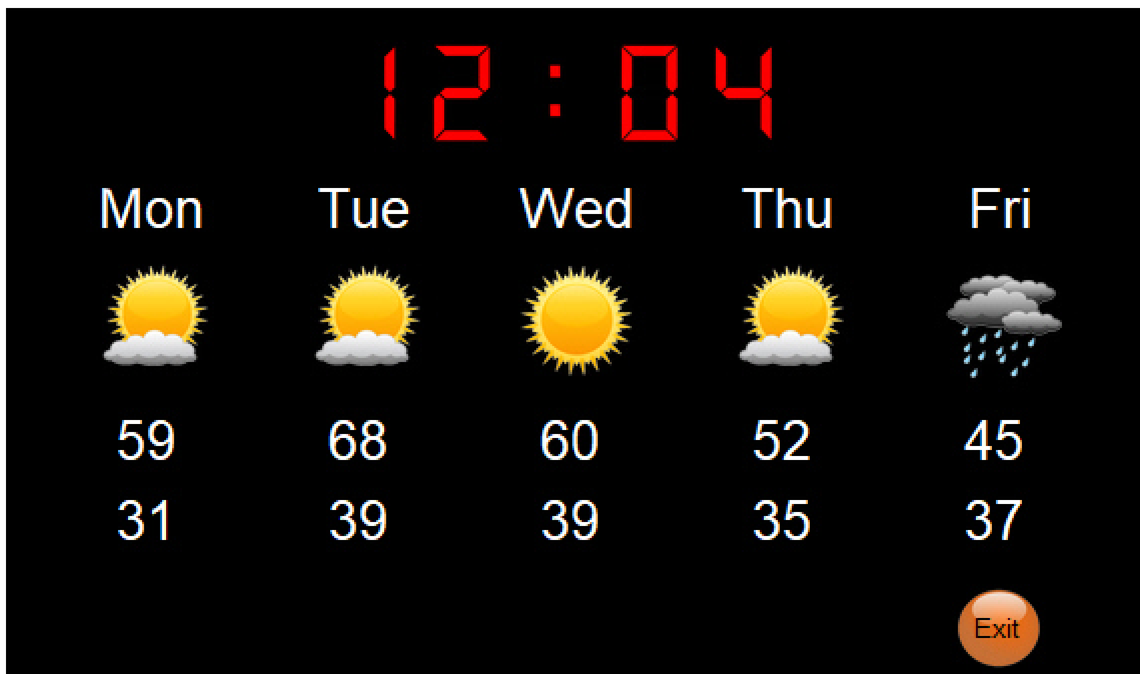


Pi Windows

Perhaps you're looking for a way to interact with your **Raspberry Pi** in a more friendly way. Your PySimpleGUI code will run on a Pi with no problem. Tkinter is alive and well on the Pi platform. Here is a selection of some of the Elements shown on the Pi. You get the same Elements on the Pi as you do Windows and Linux.



You can add custom artwork to make it look nice, like the Demo Program - Weather Forecast shown in this image:



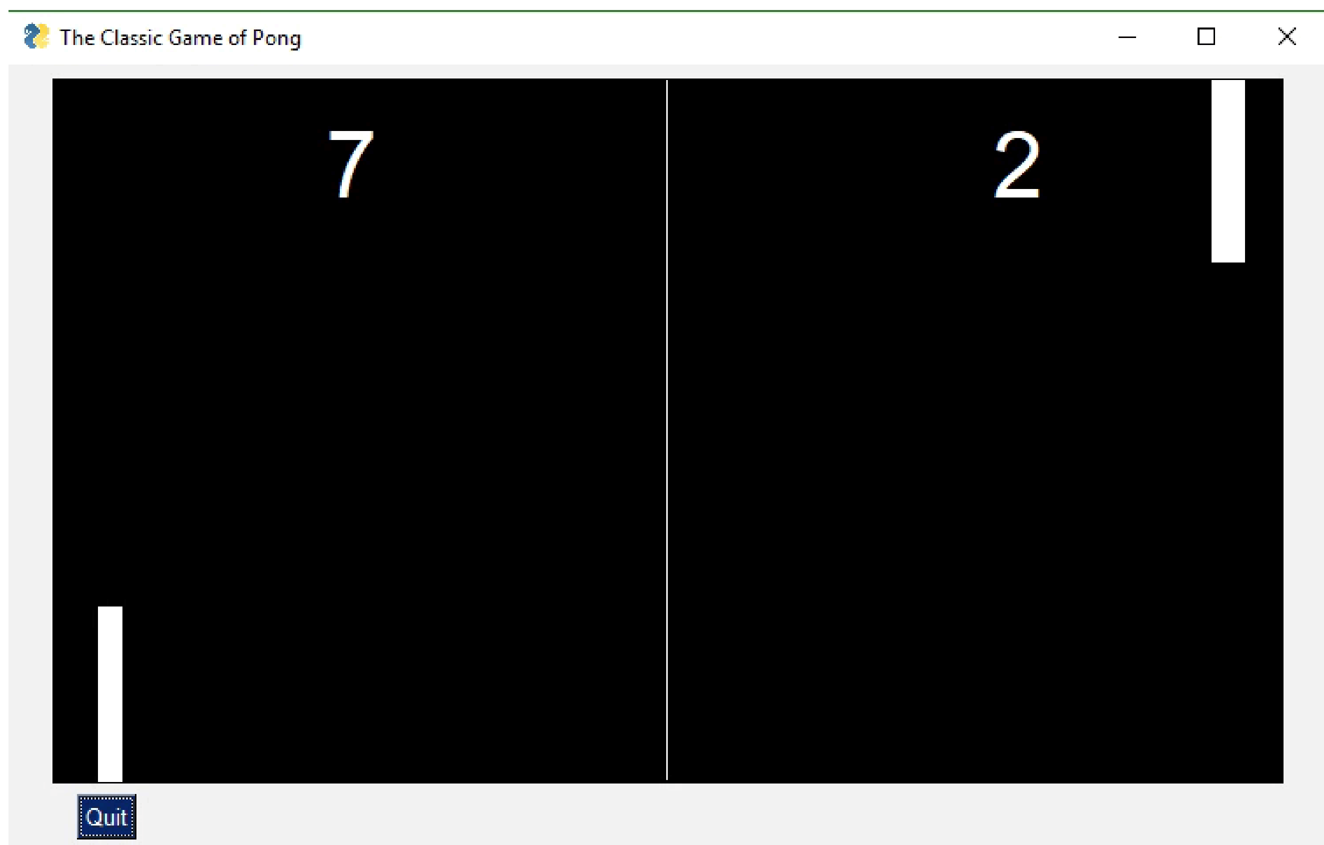
One thing to be aware of with Pi Windows, you cannot make them semi-transparent. This means that the `Window.Disappear` method will not work. Your window will not disappear. Setting the Alpha Channel will have no effect.

Don't forget that you can use custom artwork anywhere, including on the Pi. The weather application looks beautiful on the Pi. Notice there are no buttons or any of the normal looking Elements visible. It's possible to build nice looking applications, even on the lower-end platforms.

Games

It's possible to create some cool games by simply using the built-in PySimpleGUI graphic primivites like those used in this game of pong. PyGame can also be embedded into a PySimpleGUI window and code is provided to you demonstrating how. There is also a demonstration of using the pymunk physics package that can also be used for games.

Games haven't not been explored much, yet, using PySimpleGUI.



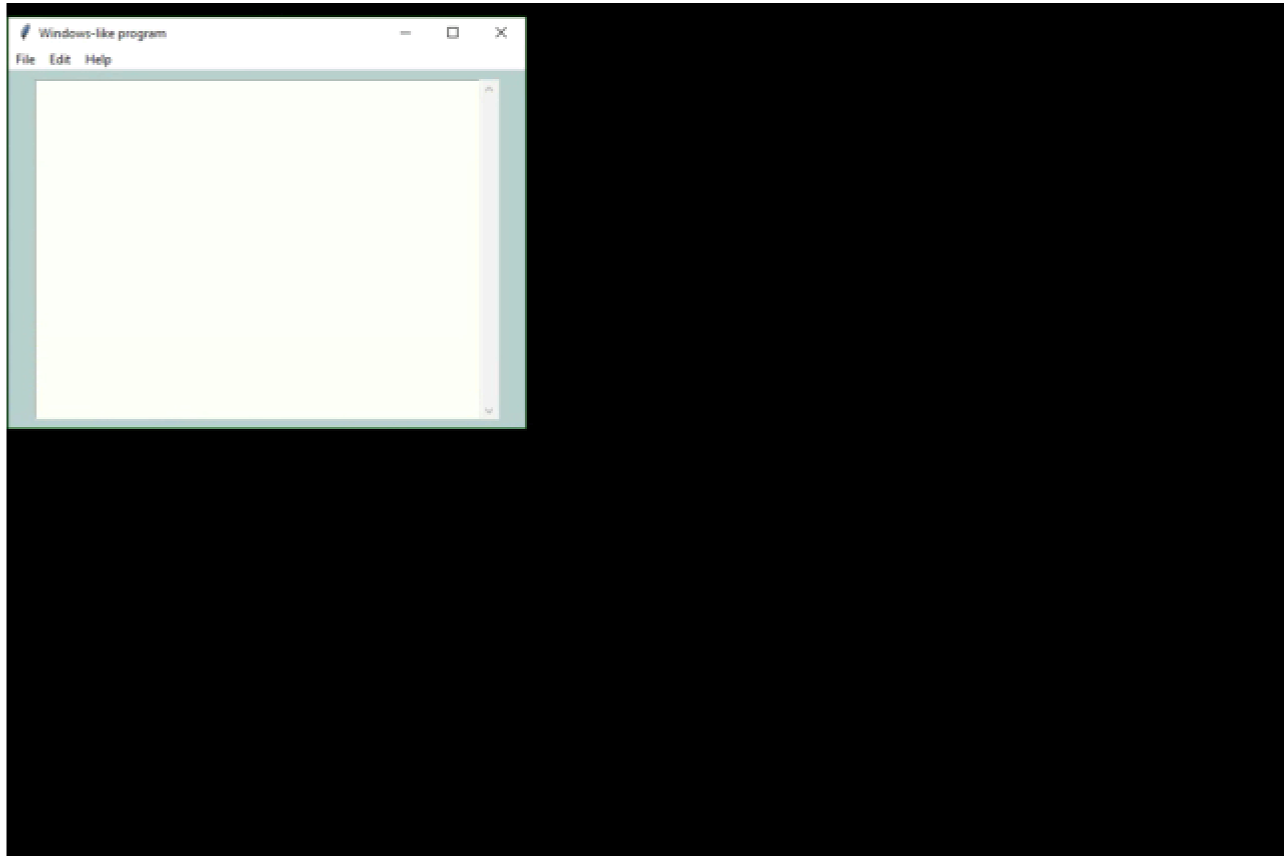
Windows Programs That Look Like Windows Programs

Do you have the desire to share your code with other people in your department, or with friends and family? Many of them may not have Python on their computer. And in the corporate environment, it may not be possible for you to install Python on their computer.

`PySimpleGUI + PyInstaller` to the rescue!!

Combining PySimpleGUI with PyInstaller creates something truly remarkable and special, a Python program that looks like a Windows WinForms application.

The application you see below with a 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. Nice, huh?



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.

There is even a PySimpleGUI program that will take your PySimpleGUI program and turn it into an EXE. It's nice because you can use a GUI to select your file and all of the output is shown in the program's window, in realtime.

Background - Why PySimpleGUI Came to Be

Feel free to skip all this if you don't care to know the backstory and reasons behind decisions.

There was a project looming and a GUI was needed. It wasn't a very complex GUI so thus began a search for a simplified GUI package that would enable me to work with tkinter easier. I found a few, and they were pretty popular too, but they lacked the full-compliment of Widgets

and it was impossible to define my own window using those widgets.

A whacky idea came to mind... what if I wrote a simplified GUI and then used THAT to write my application. It would be a lot less code and it would be "easy" to write my application then. And that is exactly what was done.

First an early version of PySimpleGUI was written that had a subset of the Elements avaiable today. It had just enough for my application. Then I wrote my application in PySimpleGUI.

Thus PySimpleGUI was born out of necessity and it's been the necessity of others that have helped evolve it into the package it is today. It would not be 1/2 as good without the help of the community.

Once PySimpleGUI was done, it was time to start working on "the ports". And, of course, also this documentation.

The Non-OO and Non-Event-Driven Model

The two "advanced concepts" that beginning Python students have with GUIs are the use of classes and callbacks with their associated communication and coordination mechanisms (semaphores, queues, etc)

How do you make a GUI interface easy enough for first WEEK Python students?

This meant classes could be used to build and use it, but classes can not be part of the code the user writes. Of course, an OO design is quite possible to use with PySimpleGUI, but it's not a ***requirement***. The sample code and docs stay away from writing new classes in the user space for the most part.

What about those pesky callbacks? They're difficult for beginners to grasp and they're a bit of a pain in the ass to deal with. The way PySimpleGUI got around events was to utilize ***a "message passing" architecture*** instead.

Instead of a user function being called when there's some event, instead the information is "passed" to the user when they call the function `Window.Read()`

Everything is returned through this `Window.Read` call. Of course the underlying GUI frameworks still perform callbacks, but they all happen inside of PySimpleGUI where they are turned into messages to pass to you.

All of the boilerplate code, the event handling, widget creation, frames containing widgets, etc, are **exactly the same** objects and calls that you would be writing if you wrote directly in tktiner, Qt, etc. With all of this code out of the way and done for you, that leaves you with the task of doing something useful with the information the user entered. THAT, afterall, is the goal here.... getting user information and acting on it.

The full complement of Widgets are available to you via PySimpleGUI Elements. And those widgets are presented to you in a unique and fun way.

If you wish to learn more about the Architecture of PySimpleGUI, take a look at the Architecture document located on ReadTheDocs.

The Result

A GUI that's appealing to a broad audience that is highly customizable, easy to program, and is solid with few bugs and rarely crashes (99% of the time it's some other error that causes a crash).

PySimpleGUI is becoming more and more popular. The number of installs and the number of successes grows daily. Pip installs have exceeded 350,000 in the first year of existance. Over 300 people a day visit the GitHub and the project has 1,800 stars (thank you aweesome users!)

The number of ports is up to 4. The number of integrations with other technologies is constantly being expanded. It's a great time to try PySimpleGUI! You've got no more than 5 or 10 minutes to lose.

Caution is needed, however, when working with the unfinished ports. PySimpleGUI, the tkinter version, is the only fully complete port. Qt is next. All of its Elements are completed, but not all of the options of each element are done. PySimpleGUIWeb is next in order of completness and then finally PySimpleGUIWx.

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 multi-windowed GUIs.

For those of you that have heard PySimpleGUI is only good for doing the most simplest of GUIs, this feature list should put that myth to rest. **The SIMPLE part of PySimpleGUI is how much effort *you* expend to write a GUI, not the complexity of the program you are able to create.** It's literally "simple" to do... and it is not limited to simple problems.

Features of PySimpleGUI include:

- Support for Python versions 2.7 and 3
- Text
- Single Line Input

- Buttons including these types:
    - File Browse
    - Files Browse
    - Folder Browse
    - SaveAs
    - Normal button that returns event
    - Close window
    - Realtime
    - Calendar chooser
    - Color chooser
    - Button Menu
- Checkboxes
- Radio Buttons
- Listbox
- Option Menu
- Menubar
- Button Menu
- Slider
- Spinner
- Dial
- Graph
- Frame with title
- Icons
- Multi-line Text Input
- Scroll-able Output
- Images
- Tables
- Trees
- Progress Bar Async/Non-Blocking Windows
- Tabbed windows
- Paned windows
- Persistent Windows
- Multiple Windows - Unlimited number of windows can be open at the same time
- 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
- Horizontal and Verticle Separators
- 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 (very classy looking)
- Always on top windows
- Menus with ALT-hotkey
- Right click pop-up menu
- Tooltips
- Clickable text
- Transparent windows
- Movable windows
- Animated GIFs
- No async programming required (no callbacks to worry about)
- Built-in debugger and REPL
- User expandable by accessing underlying GUI Framework widgets directly

---

Design Goals

With the developer being the focus, the center of it all, it was important to keep this mindset at all times, including now, today. Why is this such a big deal? Because this package was written so that the universe of Python applications can grow and can **include EVERYONE into the GUI tent.**

> Up in 5 minutes

Success #1 has to happen immediately. Installing and then running your first GUI program. FIVE minutes is the target. The Pip install is under 1 minute. Depending on your IDE and development environment, running your first piece of code could be a copy, paste, and run. This isn't a joke target; it's for real serious.

> Beginers and Advanted Together

Design an interface that both the complete beginner can understand and use that has enough depth that an advanced programmer can make some very nice looking GUIs amd not feel like they're playing with a "toy".

> Success After Success

Success after success.... this is the model that will win developer's hearts. This is what users love about PySimpleGUI. Make your development progress in a way you can run and test your code often. Add a little bit, run it, see it on your screen, smile, move on.

> Copy, Paste, Run.

The Cookbook and Demo Programs are there to fulfill this goal. First get the user seeing on their screen a working GUI that's similar in some way to what they want to create.

If you're wanting to play with OpenCV download the OpenCV Demo Programs and give them a try. Seeing your webcam running in the middle of a GUI window is quite a thrill if you're trying to integrate with the OpenCV package.

"Poof" instant running OpenCV based application == Happy Developer

> Make Simpler Than Expected Interfaces

The Single Line Progress Meter is a good example. It requires one and only 1 line of code. Printing to a debug window is as easy as replacing `print` with `sg.Print` which will route your console output to a scrolling debug window.

> Be Pythonic

Be Pythonic...

This one is difficult for me to define. The code implementing PySimpleGUI isn't PEP8 compliant, but it is consistent. The important thing was what the user saw and experienced while coding, NOT the choices for naming conventions in the implementation code. The user interface to PySimpleGUI now has a PEP8 compliant interface. The methods are snake_case now (in addition to retaining the older CamelCase names)

I ended up defining it as - attempt to use language constructs in a natural way and to exploit some of Python's interesting features. It's Python's lists and optional parameters make PySimpleGUI work smoothly.

Here are some Python-friendly aspects to PySimpleGUI:

- Windows are represented as Python lists of Elements
- Return values are an "event" such a button push and a list/dictionary of input values
- The SDK calls collapse down into a single line of Python code that presents a custom GUI and returns values should you want that extreme of a single-line soluition
- Elements are all classes. Users interact with elements using class methods but are not required to write their own classes

- Allow keys and other identifiers be any format you want. Don't limit user to particular types needlessly.
- While some disagree with the single source file, I find the benefits greatly outweigh the negatives

Lofty Goals

> Teach GUI Programming to Beginners

By and large PySimpleGUI is a "pattern based" SDK. Complete beginners can copy these standard design patterns or demo programs and modify them without necessarily understanding all of the nuts and bolts of what's happening. For example, they can modify a layout by adding elements even though they may not yet grasp the list of lists concept of layouts.

Beginners certainly can add more `if event == 'my button':` statements to the event loop that they copied from the same design pattern. They will not have to write classes to use this package.

> Capture Budding Graphic Designers & Non-Programmers

The hope is that beginners that are interested in graphic design, and are taking a Python course, will have an easy way to express themselves, right from the start of their Python experience. Even if they're not the best programmers they will be able express themselves to show custom GUI layouts, colors and artwork with ease.

> Fill the GUI Gap

There is a noticeable gap in the Python GUI solution. Fill that gap and who knows what will happen. At the moment, to make a tradiional GUI window using tkinter, Qt, WxPython and Remi, it takes much more than a week, or a month of Python education to use these GUI packages.

They are out of reach of the beginners. Often WAY out of reach. And yet, time and time again, beginners that say they JUST STARTED with Python will ask on a Forum or Reddit for a GUI pacakage recommendation. 9 times out of 10 Qt is recommended. (smacking head with hand). What a waste of characters. You might as well have just told them, "give up".

> Is There a There?

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 one attempt to find out. So far, it sure looks like there's PLENTY of demand in this area.

Getting Started with PySimpleGUI

There is a "Troubleshooting" section towards the end of this document should you run into real trouble. It goes into more detail about what you can do to help yourself.

Installing PySimpleGUI

Of course if you're installing for Qt, WxPython, Web, you'll use PySimpleGUIQt, PySimpleGUIWx, and PySimpleGUIWeb instead of straight PySimpleGUI in the instructions below. You should already have the underlying GUI Framework installed and perhaps tested. This includes tkinter, PySide2, WxPython, Remi

Installing on Python 3

`pip install --upgrade PySimpleGUI`

On some systems you need to run pip3. (Linux and Mac)

`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 PySimpleGUI`

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`

`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 you need to install `tkinter`.

For python 2.7

`sudo apt-get install python-tk`

For python 3 `sudo apt-get install python3-tk`

More information about installing tkinter can be found here: https://www.techinfected.net/2015/09/how-to-install-and-use-tkinter-in-ubuntu-debian-linux-mint.html

Installing for Python 2.7

`pip install --upgrade PySimpleGUI27` or `pip2 install --upgrade PySimpleGUI27`

You may need to also install "future" for version 2.7

`pip install future` or `pip2 install future`

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 and Troubleshooting

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.

The Quick Test

From your commant line type: `python -m PySimpleGUI`

Of course if you're on Linux/Mac and need to run using the command `python3` then of course type that.

This will display the same window as these instructions:

Instructions for Testing Python 2.7:

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

Instructions for Testing Python 3:

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

You will see a "test harness" that exercises the SDK, tells you the version number, allows you to try

Finding Out Where Your PySimpleGUI Is Coming From

It's **critical** for you to be certain where your code is coming from and which version you're running.

Sometimes when debugging, questions arise as to exactly which PySimpleGUI you are running. The quick way to find this out is to again, run Python from the command line. This time you'll type:

```
>>> import PySimpleGUI as sg
>>> sg
```

When you type sg, Python will tell you the full patch to your PySimpleGUI file / package. This is critical information to know when debugging because it's really easy to forget you've got an old copy of PySimpleGUI laying around somewhere.

Finding Out Where Your PySimpleGUI Is Coming From (from within your code)

If you continue to have troubles with getting the right version of PySimpleGUI loaded, THE ***definitive*** way to determine where your program is getting PySimpleGUI from is to add a print to your program. It's that *simple*! You can also get the version you are running by also printing

```
import PySimpleGUI as sg

print(sg)
print(sg.version)
```

Just like when using the REPL >>> to determine the location, this `print` in your code will display the same path information.

Manual installation

If you're not connected to the net on your target machine, or pip isn't working, or you want to run the latest code from GitHub, then all you have to do is place the single PySimpleGUI source file `PySimpleGUI.py` (for tkinter port) and place it in your application's folder (the folder where the py file is that imports PySimpleGUI). Your application will load that local copy of PySimpleGUI as if it were a package.

Be ***sure*** that you delete this PySimpleGUI.py file if you install a newer pip version. Often the sequence of events is that a bug you've reported was fixed and checked into GitHub. You download the PySimpleGUI.py file (or the appropriately named one for your port) and put with

your app. Then later your fix is posted with a new release on PyPI. You'll want to delete the GitHub one before you install from pip.

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 document.

IDEs

A lot of people ask about IDEs, and many outright fear PyCharm. Listen up.... compared to your journey of learning Python, learning to use PyCharm as your IDE is NOTHING. It's a DAY typically (from 1 to 8 hours). Or, if you're really really new, perhaps as much as a week *to get used to*. So, we're not talking about you needing to learn to flap your arms and fly.

To sum up that paragraph, stop whining like a little b*tch. You're a grown man/woman, act like it. "But it's hard..." If you found this package, then you're a bright person :-) Have some confidence in yourself for Christ sake.... I do. Not going to lead you off some cliff, promise!

Some IDEs provide virtual environments, but it's optional. PyCharm is one example. For these, you will either use their GUI interface to add packages or use their built-in terminal to do pip installs. **It's not recommended for beginners to be working with Virtual Environments.** They can be quite confusing. However, if you are a seasoned professional developer and know what you're doing, there is nothing about PySimpleGUI that will prevent you from working this way. It's mostly a caution for beginners because more often than not, they get really messed up and confused.

Officially Supported IDEs

A number of IDEs have **known problems with PySimpleGUI**. IDLE, Spyder, and Thonny all have known, demonstrable, problems with intermittent or inconsistent results, **especially when a program exits** and you want to continue to work with it. ***Any IDE that is based on tkinter is going to have issues with the straight PySimpleGUI port.*** This is NOT a PySimpleGUI problem.

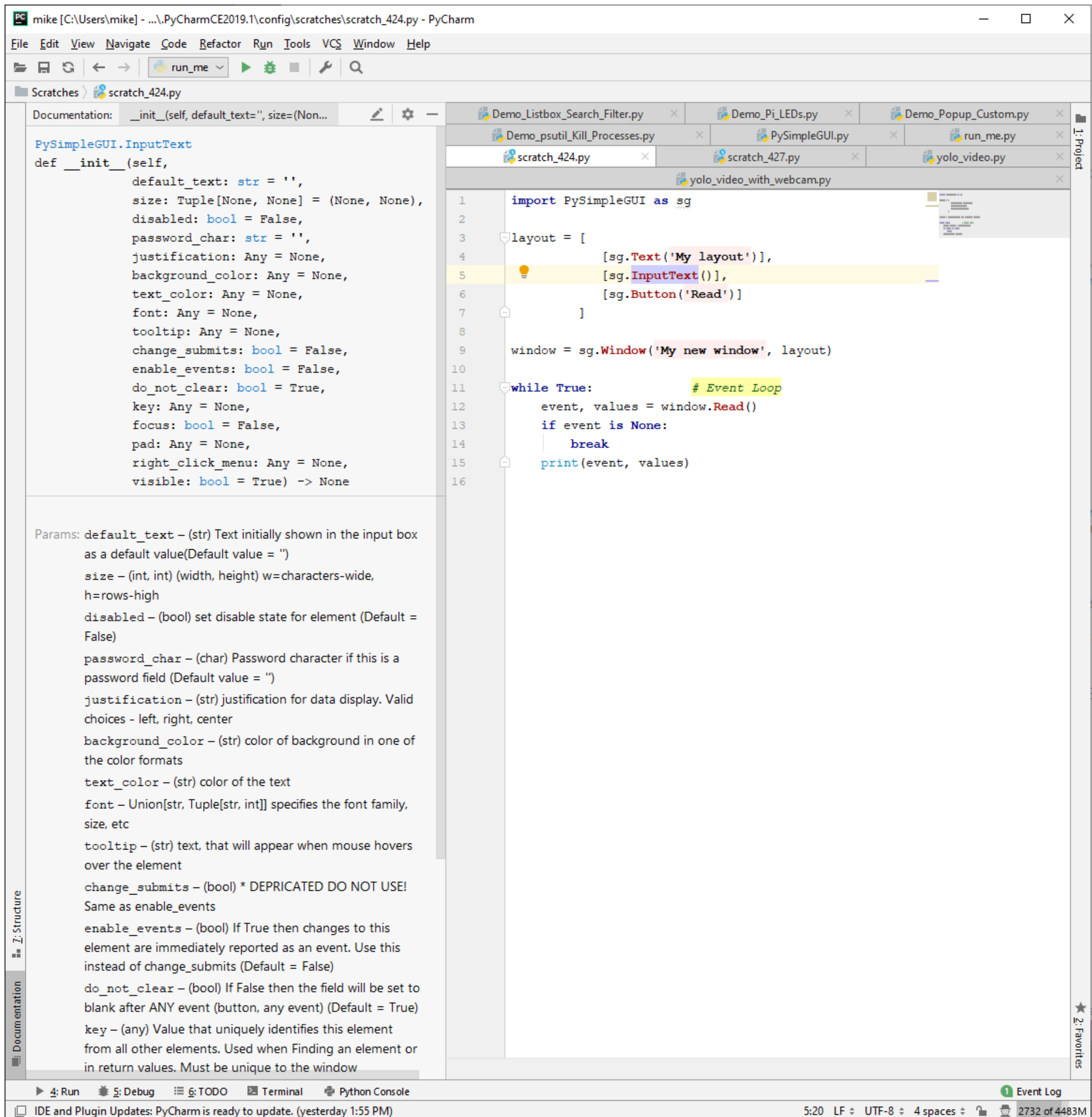
The official list of supported IDEs is: 1. PyCharm (or course this is THE IDE to use for use with PySimpleGUI) 2. Wing 3. Visual Studio

If you're on a Raspberry Pi or some other limited environment, then you'll may have to use IDLE or Thonny. Just be aware there could be problems using the debugger to debug due to both using tkinter.

Using The Docstrings (Don't skip this section)

Beginning with the 4.0 release of PySimpleGUI, the tkinter port, a whole new world opened up for PySimpleGUI programmers, one where referencing the readme and ReadTheDocs documentation is no longer needed. PyCharm and Wing both support these docstrings REALLY well and I'm sure Visual Studio does too. Why is this important? Because it will teach you the PySimpleGUI SDK as you use the package.

Don't know the parameters and various options for the `InputText` Element? It's a piece of cake with PyCharm. You can set PyCharm to automatically display documentation about the class, function, method, etc, that your cursor is currently sitting on. You can also manually bring up the documentation by pressing CONTROL+Q. When you do, you'll be treated to a window similar to this:



Note that my cursor is on `InputText`. On the left side of the screen, the `InputText` element's parameters are not just shown to you, but they are each individually described to you, and, the type is shown as well. *I mean, honestly, how much more could you ask for?*

OK, I suppose you could ask for a smaller window that just shows the parameters are you're typing them in. Well, OK, in PyCharm, when your cursor is between the `()` press CONTROL+P. When you do, you'll be treated to a little window like this one:

```
self: InputText, default_text: str='', size: Tuple[None, None]=(None, None),
disabled: bool=False, password_char: str='', justification=None,
background_color=None, text_color=None, font=None,
tooltip=None, change_submits: bool=False, enable_events: bool=False,
do_not_clear: bool=True, key=None, focus: bool=False,
pad=None, right_click_menu=None, visible: bool=True

            [sg.Text('My layout')],
            [sg.InputText(defa)],
            [sg.Button('Read')]
        ]
```

See.... written with the "Developer" in mind, at all times. It's about YOU, Mr/Ms Developer! So enjoy your package.

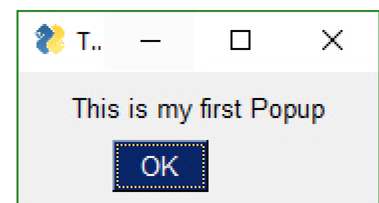
The other ports of PySimpleGUI (Qt, WxPython, Web) have not yet had their docstrings updated. They're NEXT in line to be better documented. Work on a tool has already begun to make that happen sooner than later.

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.

WARNING Do NOT use PySimpleGUI with Python 3.7.3 and 3.7.4. tkiter is having issues with that release. Things like Table colors stopped working entirely. None of us want to debug tkinter code. It's difficult enough debugging your code and PySimpleGUI code. A lot of time has already been spent debugging this one so no need for you to suffer too.

Python 3.7

It puzzles me why a beginner would install 3.7. Or even a seasoned programmer. What specific feature of 3.7 are you using that is not in 3.6? If you are unable to answer this, then ***it's strongly suggested that you run 3.6***, an immensely solid release of Python with all those goodie inside like f-strings. If you must run 3.7, try 3.7.2 instead. It does work with PySimpleGUI with no known issues.

Using - Python 2.7

Those using Python 2.7 will import a different module name

```
import PySimpleGUI27 as sg
```

Code to Automatically Import Correct Version

Many of the demo programs use this check to see which package to use:

```
import sys
if sys.version_info[0] >= 3:
    import PySimpleGUI as sg
else:
    import PySimpleGUI27 as sg
```

This will automatically import the correct library based on the Python version number reported by the Python interpreter.

NOTE: It's 2019 and 2.7 support is being systematically removed. This construct will be removed from the demo programs shortly. 2.7 users can still run these demos, but they will need to change the import from PySimpleGUI to PySimpleGUI27. It save 4 lines of code and an import from sys in the process.

PEP8 Bindings For Methods and Functions

Beginning with release 4.3 of PySimpleGUI, ***all methods and function calls*** have PEP8 equivalents. This capability is only available, for the moment, on the PySimpleGUI tkinter port. It is being added, as quickly as possible, to all of the ports.

As long as you know you're sticking with tkinter for the short term, it's safe to use the new bindings.

The Non-PEP8 Methods and Functions

Why the need for these bindings? Simply put, the PySimpleGUI SDK has a PEP8 violation in the method and function names. PySimpleGUI uses CamelCase names for methods and functions. PEP8 suggests using snake_case_variables instead.

This has not caused any problems and few complaints, but it's important the the interfaces into PySimpleGUI be compliant. Perhaps one of the reasons for lack of complaints is that the Qt library also uses SnakeCase for its methods. This practice has the effect of labelling a

package as being "not Pythonic" and also suggests that ths package was originally used in another language and then ported to Python. This is exactly the situation with Qt. It was written for C++ and the interfaces continue to use C++ conventions.

PySimpleGUI was written in Python, for Python. The reason for the name problem was one of ignorance. The PEP8 convention wasn't understood by the developers when PySimpleGUI was designed and implemented.

You can, and will be able to for some time, use both names. However, at some point in the future, the CamelCase names will disappear. A utility is planned to do the conversion for the developer when the old names are remove from PySimpleGUI.

The help system will work with both names as will your IDE's docstring viewing. However, the result found will show the CamelCase names. For example `help(sg.Window.read)` will show the CamelCase name of the method/function. This is what will be returned:

`Read(self, timeout=None, timeout_key='__TIMEOUT__')`

The Renaming Convention

To convert a CamelCase method/function name to snake_case, you simply place an `_` where the Upper Case letter is located. If there are none, then only the first letter is changed.

`Window.FindElement` becomes `Window.find_element`

Class Variables

For the time being, class variables will remain the way they are currently. It is unusual, in PySimpleGUI, for class variables to be modified or read by the user code so the impact of leaving them is rarely seen in your code.

"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.

Think of Popups as your first windows, sorta like your first bicycle. It worked well, but was limited. It probably wasn't long before you wanted more features and it seemed too limiting for your newly found sense of adventure.

When you've reached the point with Popups that you are thinking of filing a GitHub "Enhancement Issue" to get the Popup call extended to include a new feature that you think would be helpful.... not just to you but others is what you had in mind, right? For the good of others.

It's at THIS time that you should immediately turn to the section entitled "Custom Window API Calls - Your First Window". Congratulations, you just graduated and are not an official "GUI Designer". Oh, nevermind that you only started learning Python 2 weeks ago, you're a real GUI Designer now so buck up and start acting like one.

But, for now, let's stick with these 1-line window calls, the Popups.

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`.

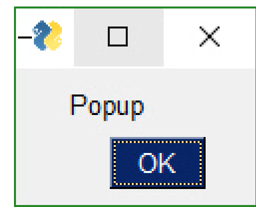
The function `PopupTimed` or `PopupAutoClose` are popup windows that will automatically close after come period of time.

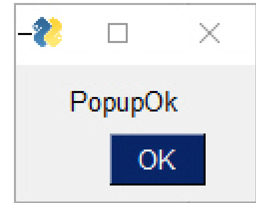
Here is a quick-reference showing how the Popup calls look.

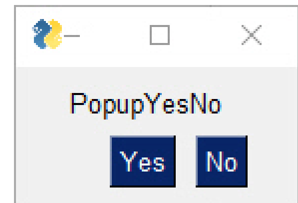
```
sg.Popup('Popup')
sg.PopupOk('PopupOk')
sg.PopupYesNo('PopupYesNo')
sg.PopupCancel('PopupCancel')
sg.PopupOKCancel('PopupOKCancel')
sg.PopupError('PopupError')
sg.PopupTimed('PopupTimed')
sg.PopupAutoClose('PopupAutoClose')
```

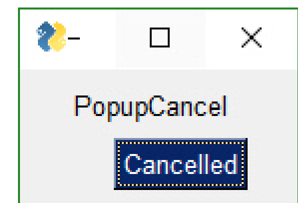
Preview of popups:

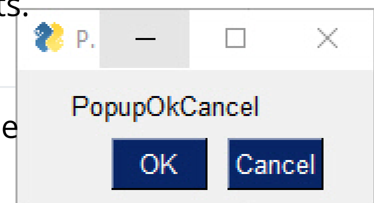
Popup - Display a popup Window with as many parms as you wish to include. This is the GUI equivalent of the "print" statement. It's also great for "pausing" your program's flow until the user can read some error messages.

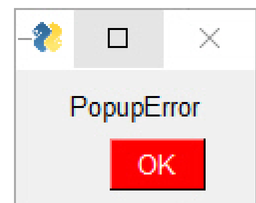


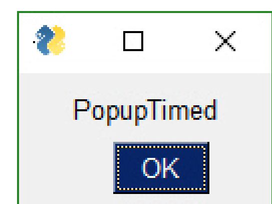













```
Popup(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    button_type=0,
    auto_close=False,
    auto_close_duration=None,
    custom_text=(None, None),
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	(Any) Variable number of your arguments. Load up the call with stuff to see!
title	(str) Optional title for the window. If none provided, the first arg will be used instead.
button_color	Tuple[str, str] Color of the buttons shown (text color, button color)
background_color	(str) Window's background color
text_color	(str) text color
button_type	(enum) NOT USER SET! Determines which pre-defined buttons will be shown (Default value = POPUP_BUTTONS_OK). There are many Popup functions and they call Popup, changing this parameter to get the desired effect.

Name	Meaning
auto_close	(bool) If True the window will automatically close
auto_close_duration	(int) time in seconds to keep window open before closing it automatically
custom_text	Union[Tuple[str, str], str] A string or pair of strings that contain the text to display on the buttons
non_blocking	(bool) If True then will immediately return from the function without waiting for the user's input.
icon	Union[str, bytes] icon to display on the window. Same format as a Window call
line_width	(int) Width of lines in characters. Defaults to MESSAGE_BOX_LINE_WIDTH
font	Union[str, tuple(font name, size, modifiers) specifies the font family, size, etc
no_titlebar	(bool) If True will not show the frame around the window and the titlebar across the top
grab_anywhere	(bool) If True can grab anywhere to move the window. If no_titlebar is True, grab_anywhere should likely be enabled too
location	Tuple[int, int] Location on screen to display the top left corner of window. Defaults to window centered on screen
return	Union[str, None] Returns text of the button that was pressed. None will be returned if user closed window with X

The other output Popups are variations on parameters. Usually the button_type parameter is

the primary one changed.

The other output Popups are variations on parameters. Usually the button_type parameter is the primary one changed.

The choices for button_type are:

```
POPUP_BUTTONS_YES_NO
POPUP_BUTTONS_CANCELLED
POPUP_BUTTONS_ERROR
POPUP_BUTTONS_OK_CANCEL
POPUP_BUTTONS_OK
POPUP_BUTTONS_NO_BUTTONS
```

Note that you should not call Popup yourself with different button_types. Rely on the Popup function named that sets that value for you. For example PopupYesNo will set the button type to POPUP_BUTTONS_YES_NO for you.

Scrolled Output

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

Show a scrolled Popup window containing the user's text that was supplied. Use with as many items to print as you want, just like a print statement.

```
PopupScrolled(args,
    title=None,
    button_color=None,
    yes_no=False,
    auto_close=False,
    auto_close_duration=None,
    size=(None, None),
    location=(None, None),
    non_blocking=False)
```

Parameter Descriptions:

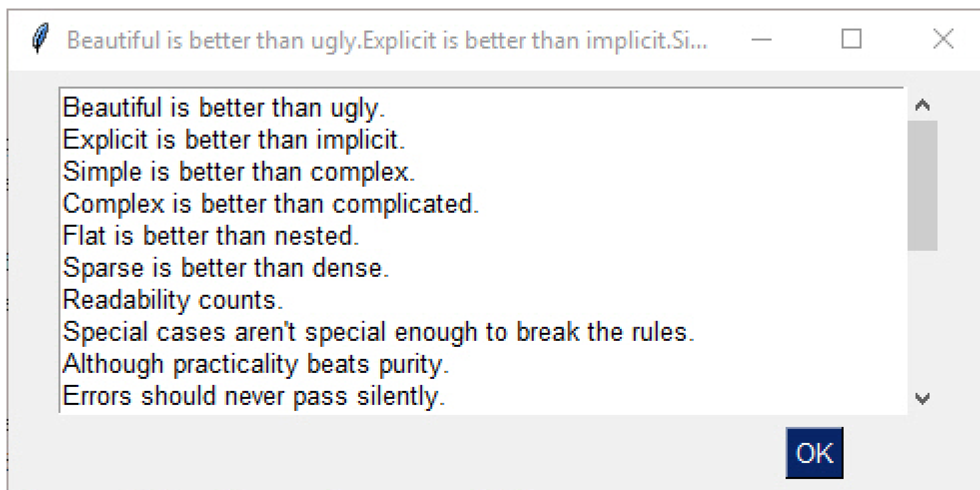
Name	Meaning
*args	(Any) Variable number of items to display
title	(str) Title to display in the window.
button_color	Tuple[str, str] button color (foreground, background)
yes_no	(bool) If True, displays Yes and No buttons instead of Ok
auto_close	(bool) if True window will close itself

Name	Meaning
auto_close_duration	Union[int, float] Older versions only accept int. Time in seconds until window will close
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
location	Tuple[int, int] Location on the screen to place the upper left corner of the window
non_blocking	(bool) if True the call will immediately return rather than waiting on user input
return	Union[str, None, TIMEOUT_KEY] Returns text of the button that was pressed. None will be returned if user closed window with X

```
PopupScrolled(*args, button_color=None, yes_no=False, auto_close=False, auto_close_duration=None, size=(None, None), location=(None, None), title=None, non_blocking=False)
```

Typical usage:

```
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.

If `non_blocking` parameter is set, then the call will not blocking waiting for the user to close the window. Execution will immediately return to the user. Handy when you want to dump out debug info without disrupting the program flow.

Show Popup window and immediately return (does not block)

```
PopupNoWait(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = True)
icon	Icon to display
line_width	Width of lines in characters

Name	Meaning
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

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. There are no return values on one of these Popups.

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. There are 3 of these input Popups to choose from, each with settings enabling customization. - `PopupGetText` - get a single line of text - `PopupGetFile` - get a filename - `PopupGetFolder` - get a folder name

Use these Popups instead of making a custom window to get one data value, call the Popup input function to get the item from the user. If you find the parameters are unable to create the kind of window you are looking for, then it's time for you to create your own window.

Use this Popup to get a line of text from the user.

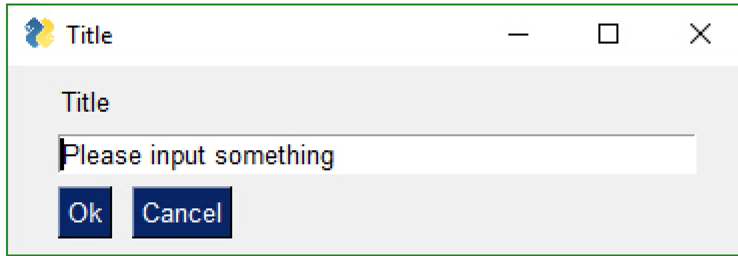
Display Popup with text entry field. Returns the text entered or None if closed / cancelled

```
PopupGetText(message,
    title=None,
    default_text="",
    password_char="",
    size=(None, None),
    button_color=None,
    background_color=None,
    text_color=None,
    icon=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

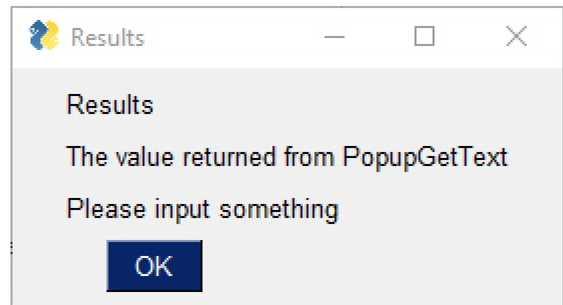
Parameter Descriptions:

Name	Meaning
message	(str) message displayed to user
title	(str) Window title
default_text	(str) default value to put into input area
password_char	(str) character to be shown instead of actually typed characters
size	Tuple[int, int] (width, height) of the InputText Element
button_color	Tuple[str, str] Color of the button (text, background)
background_color	(str) background color of the entire window
text_color	(str) color of the message text
icon	Union[bytes, str] filename or base64 string to be used for the window's icon
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
no_titlebar	(bool) If True no titlebar will be shown
grab_anywhere	(bool) If True can click and drag anywhere in the window to move the window
keep_on_top	(bool) If True the window will remain above all current windows
location	Tuyple[int, int] (x,y) Location on screen to display the upper left corner of window
return	Union[str, None] Text entered or None if window was closed or cancel button clicked

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



Gets a filename from the user. There are options to configure the type of dialog box to show. Normally an "Open File" dialog box is shown.



Display popup window with text entry field and browse button so that a file can be chosen by user.

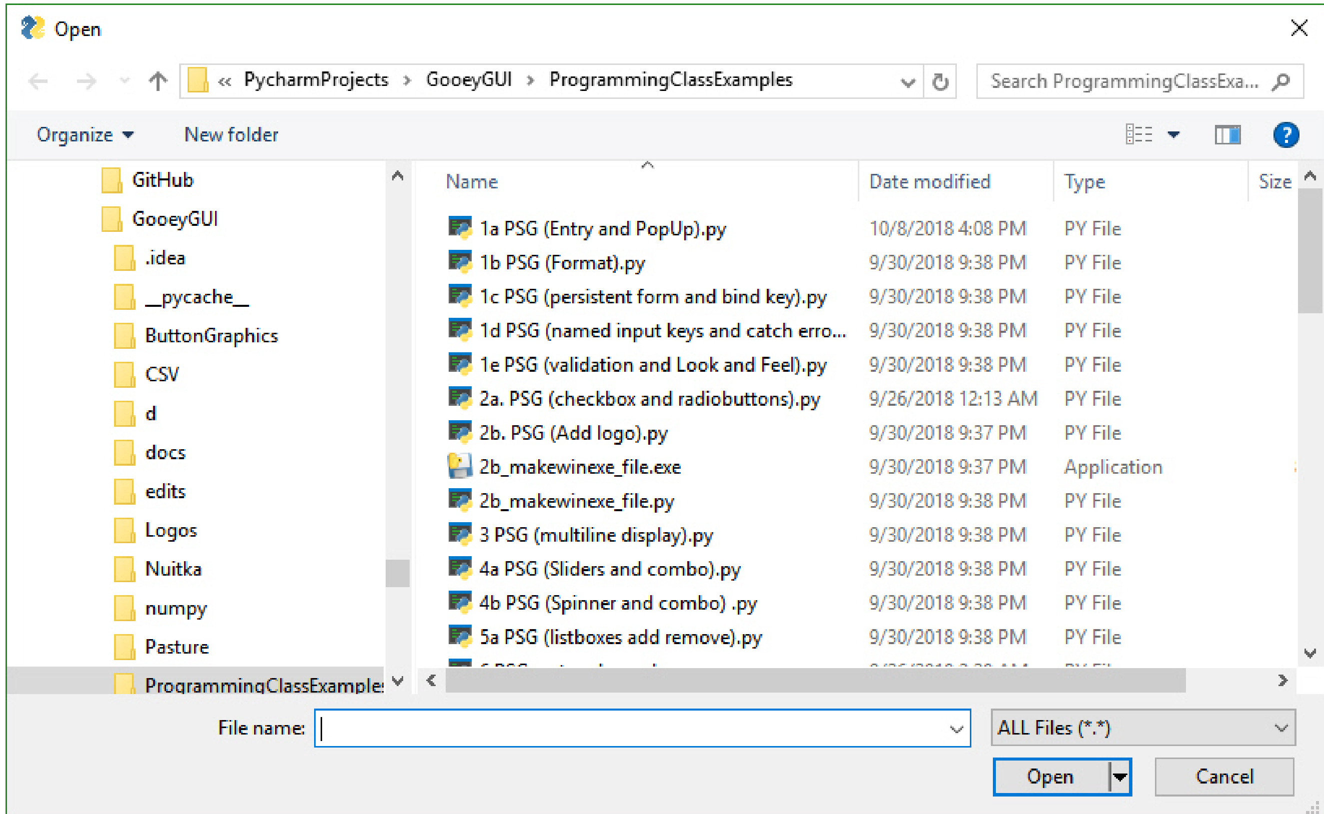
```
PopupGetFile(message,
    title=None,
    default_path="",
    default_extension="",
    save_as=False,
    multiple_files=False,
    file_types=(('ALL Files', '*.*'),),
    no_window=False,
    size=(None, None),
    button_color=None,
    background_color=None,
    text_color=None,
    icon=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None),
    initial_folder=None)
```

Parameter Descriptions:

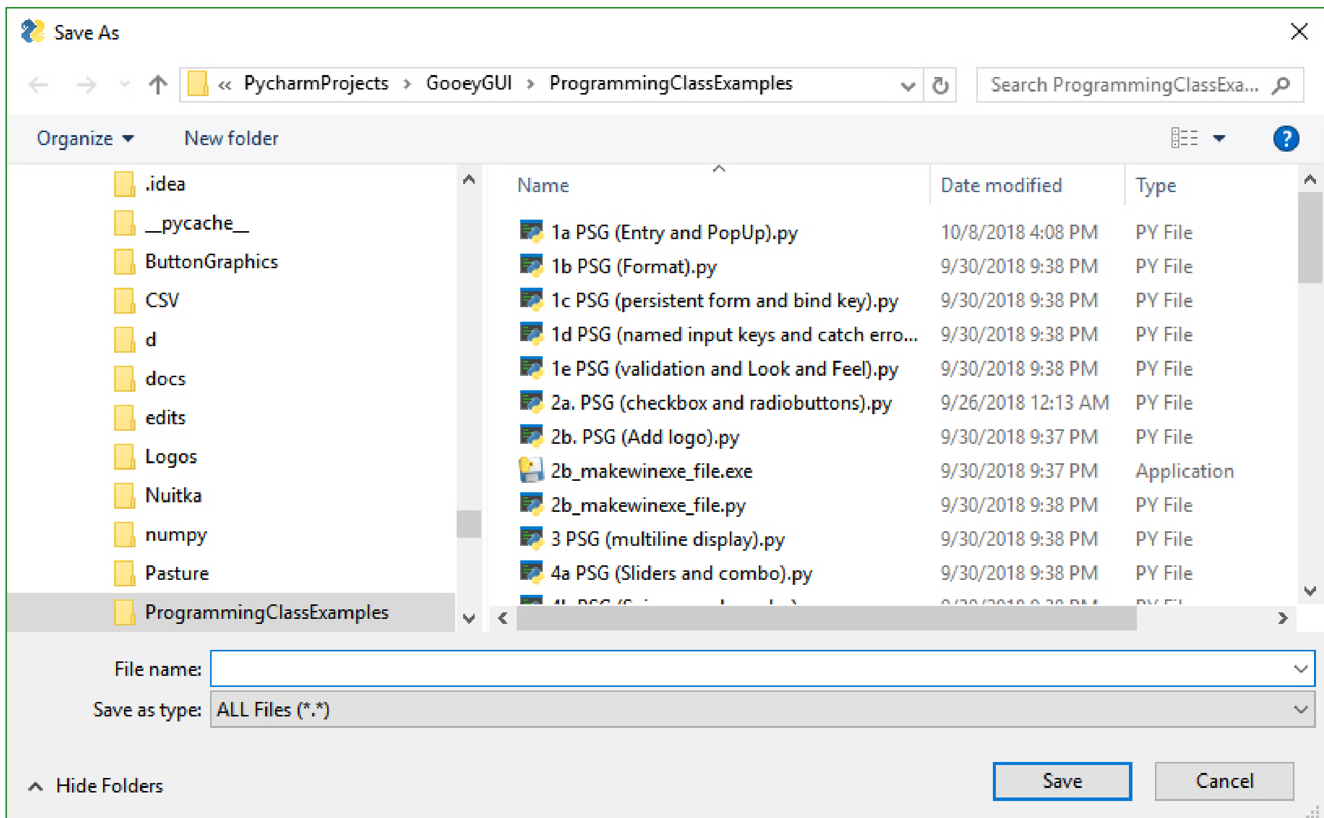
Name	Meaning
message	(str) message displayed to user
title	(str) Window title
default_path	(str) path to display to user as starting point (filled into the input field)
default_extension	(str) If no extension entered by user, add this to filename (only used in saveas dialogs)

Name	Meaning
save_as	(bool) if True, the "save as" dialog is shown which will verify before overwriting
multiple_files	(bool) if True, then allows multiple files to be selected that are returned with ';' between each filename
file_types	Tuple[Tuple[str,str]] List of extensions to show using wildcards. All files (the default) = (("ALL Files", "."),)
no_window	(bool) if True, no PySimpleGUI window will be shown. Instead just the tkinter dialog is shown
size	Tuple[int, int] (width, height) of the InputText Element
button_color	Tuple[str, str] Color of the button (text, background)
background_color	(str) background color of the entire window
text_color	(str) color of the message text
icon	Union[bytes, str] filename or base64 string to be used for the window's icon
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
no_titlebar	(bool) If True no titlebar will be shown
grab_anywhere	(bool) If True can click and drag anywhere in the window to move the window
keep_on_top	(bool) If True the window will remain above all current windows
location	Tuyple[int, int] (x,y) Location on screen to display the upper left corner of window
initial_folder	(str) location in filesystem to begin browsing
return	Union[str, None] string representing the file(s) chosen, None if cancelled or window closed with X

If configured as an Open File Popup then (save_as is not True) the dialog box will look like this.



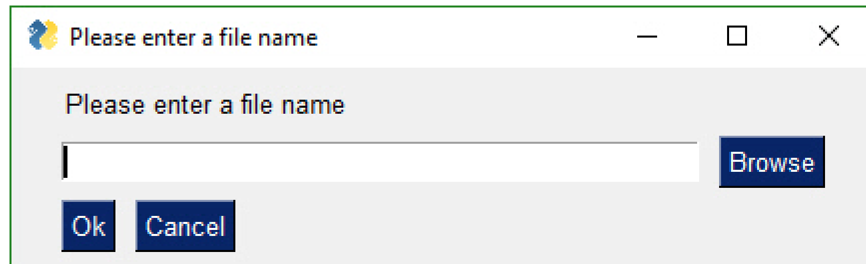
If you set the parameter save_As to True, then the dialog box looks like this:



If you choose a filename that already exists, you'll get a warning popup box asking if it's OK. You can also specify a file that doesn't exist. With an "Open" dialog box you cannot choose a non-existing file.

A typical call produces this window.

```
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.

Display popup with text entry field and browse button so that a folder can be chosen.

```
PopupGetFolder(message,
    title=None,
    default_path="",
    no_window=False,
    size=(None, None),
    button_color=None,
    background_color=None,
    text_color=None,
    icon=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None),
    initial_folder=None)
```

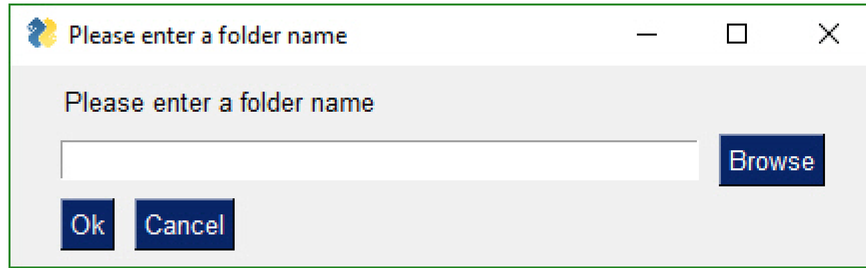
Parameter Descriptions:

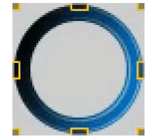
Name	Meaning
message	(str) message displayed to user
title	(str) Window title

Name	Meaning
default_path	(str) path to display to user as starting point (filled into the input field)
no_window	(bool) if True, no PySimpleGUI window will be shown. Instead just the tkinter dialog is shown
size	Tuple[int, int] (width, height) of the InputText Element
button_color	Tuple[str, str] Color of the button (text, background)
background_color	(str) background color of the entire window
text_color	(str) color of the message text
icon	Union[bytes, str] filename or base64 string to be used for the window's icon
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
no_titlebar	(bool) If True no titlebar will be shown
grab_anywhere	(bool) If True can click and drag anywhere in the window to move the window
keep_on_top	(bool) If True the window will remain above all current windows
location	Tuyple[int, int] (x,y) Location on screen to display the upper left corner of window
initial_folder	(str) location in filesystem to begin browsing
return	Union[str, None] string representing the path chosen, None if cancelled or window closed with X

This is a typpical call

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





The animated Popup enables you to easily display a "loading" style animation specified through a GIF file that is either stored in a file or a base64 variable.

Show animation one frame at a time. This function has its own internal clocking meaning you can call it at any frequency and the rate the frames of video is shown remains constant. Maybe your frames update every 30 ms but your event loop is running every 10 ms. You don't have to worry about delaying, just call it every time through the loop.

```
PopupAnimated(image_source,
    message=None,
    background_color=None,
    text_color=None,
    font=None,
    no_titlebar=True,
    grab_anywhere=True,
    keep_on_top=True,
    location=(None, None),
    alpha_channel=None,
    time_between_frames=0,
    transparent_color=None)
```

Parameter Descriptions:

Name	Meaning
image_source	Union[str, bytes] Either a filename or a base64 string.
message	(str) An optional message to be shown with the animation
background_color	(str) color of background
text_color	(str) color of the text
font	Union[str, tuple) specifies the font family, size, etc
no_titlebar	(bool) If True then the titlebar and window frame will not be shown

Name	Meaning
grab_anywhere	(bool) If True then you can move the window just clicking anywhere on window, hold and drag
keep_on_top	(bool) If True then Window will remain on top of all other windows currently shownn
location	(int, int) (x,y) location on the screen to place the top left corner of your window. Default is to center on screen
alpha_channel	(float) Window transparency 0 = invisible 1 = completely visible. Values between are see through
time_between_frames	(int) Amount of time in milliseconds between each frame
transparent_color	(str) This color will be completely see-through in your window. Can even click through

To close animated popups, call PopupAnimated with `image_source=None`. This will close all of the currently open PopupAnimated windows.

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):
```

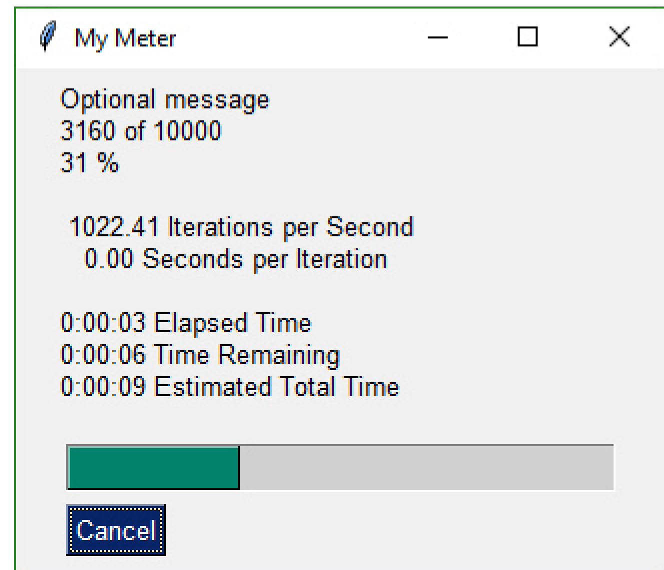
Here's the one-line Progress Meter in action!

```
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 (EasyPrint = Print = eprint)

Another call in the 'Easy' families of APIs is `EasyPrint`. As is with other commonly used PySimpleGUI calls, there are other names for the same call. You can use `Print` or `eprint` in addition to `EasyPrint`. They all do the same thing, 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 an 'sg.Print' call in your code. You can even replace your 'print' calls with calls to EasyPrint by simply sticking the statement

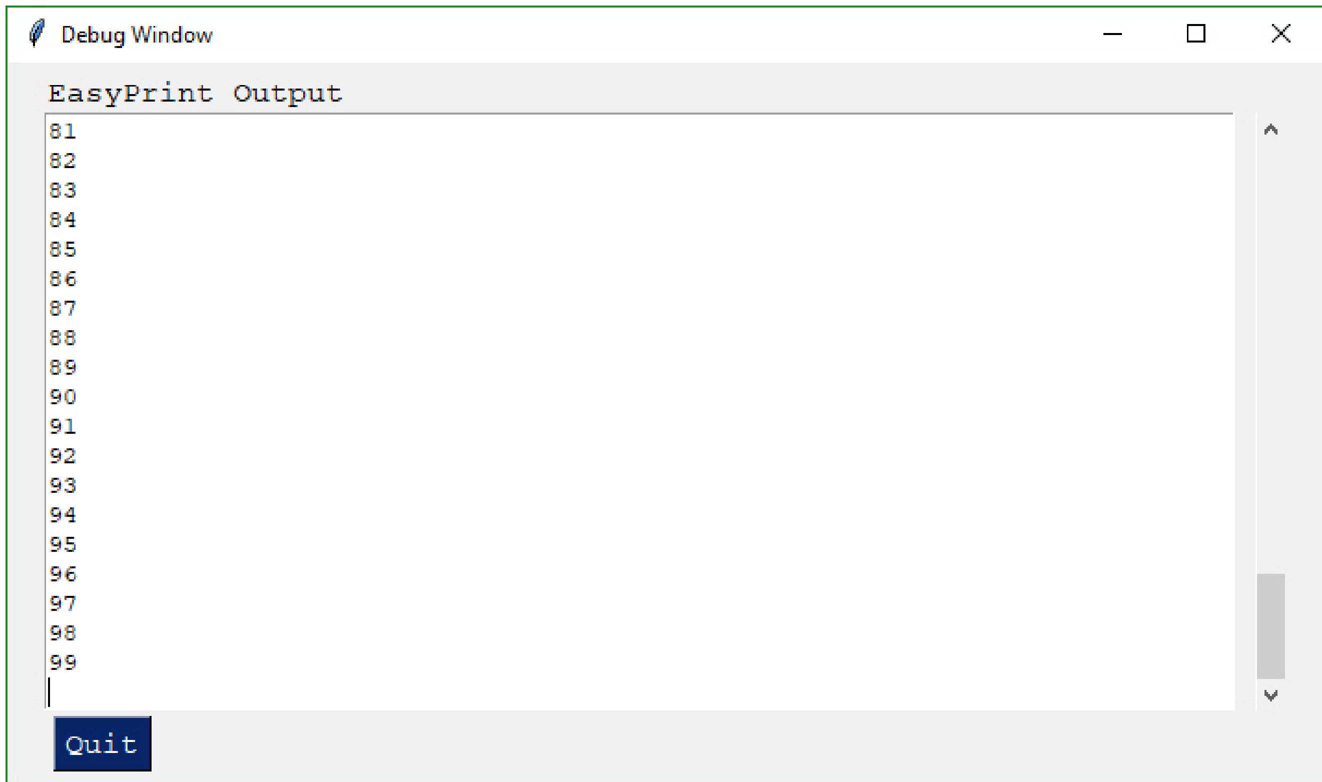
```
print = sg.EasyPrint
```

at the top of your code.

`Print` is one of the better ones to use as it's easy to remember. It is simply `print` with a capital P. `sg.Print('this will go to the debug window')`

```
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` .

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

There is an option to tell PySimpleGUI to reroute all of your stdout and stderr output to this window. To do so call EasyPrint with the parameter `do_not_reroute_stdout` set to `False` . After calling it once with this parameter set to True, all future calls to a normal `print` will go to the debug window.

If you close the debug window it will re-open the next time you Print to it. If you wish to close the window using your code, then you can call either `EasyPrintClose()` or `PrintClose()`

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-persistent 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 with an X.

Two other types of windows exist. 1. Persistent window - the `Window.read()` method returns and the window continues to be visible. This is good for applications like a chat window or a timer or anything that stays active on the screen for a while. 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. You can spot them easily as they will have a `timeout` parameter on the call to read. `event, values = window.Read(timeout=100)`

It's both not enjoyable nor helpful to immediately jump into tweaking each and every little thing available to you. Make some simple windows. Use the Cookbook and the Demo Programs as a way to learn and as a "starting point".

The window Designer

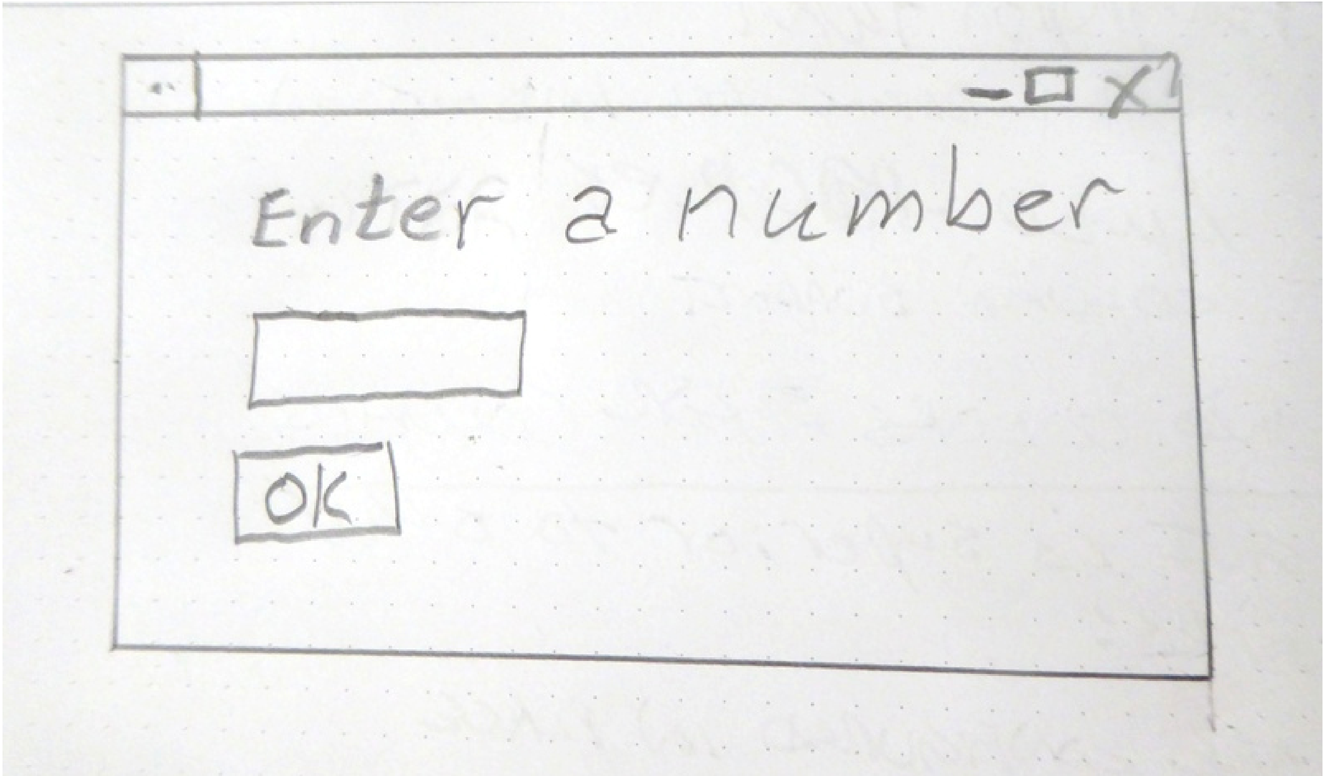
The good news to newcomers to GUI programming is that PySimpleGUI has a window designer. Better yet, the window designer requires no training, no downloads, and everyone knows how to use it.

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

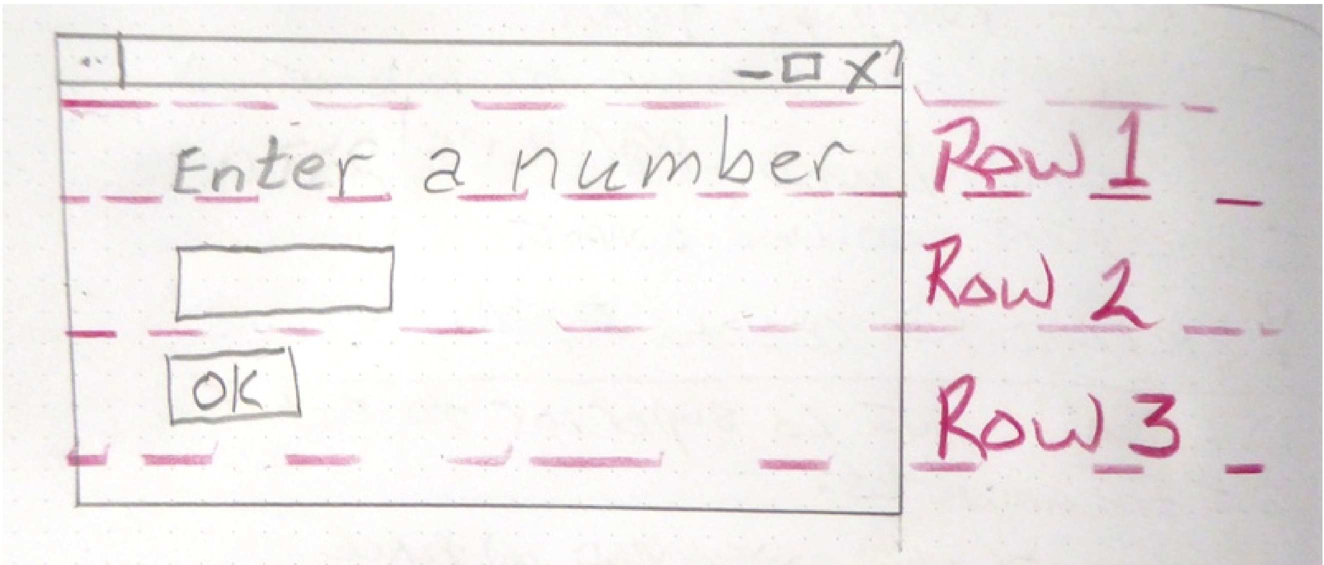
Let's take a couple of examples.

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.

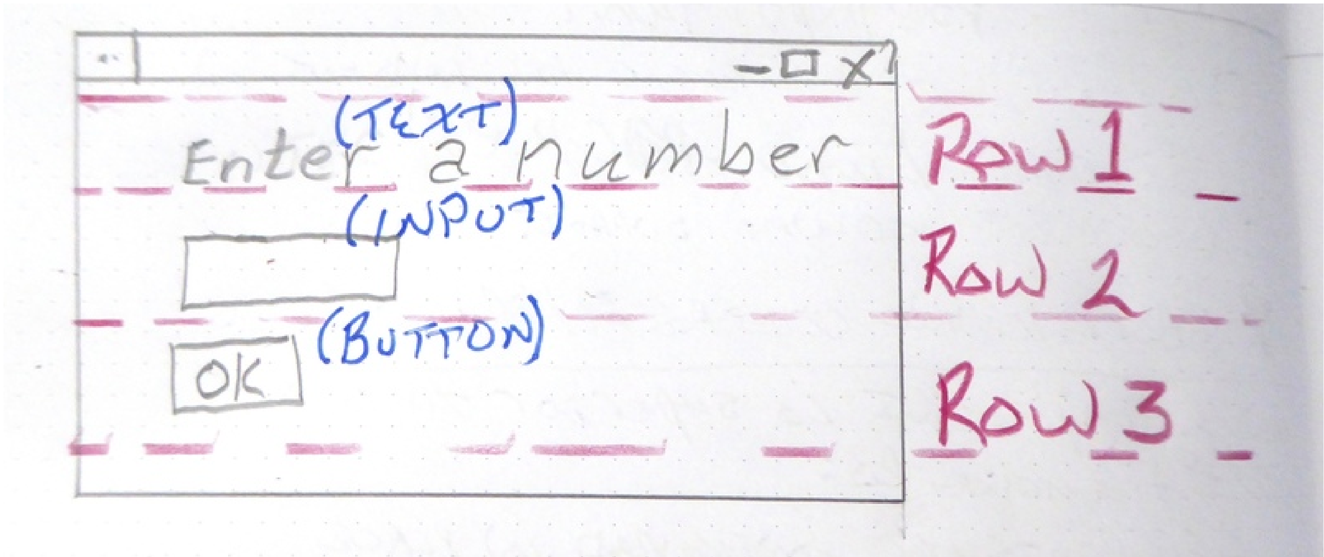
Step 1- Sketch the GUI



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()] ]

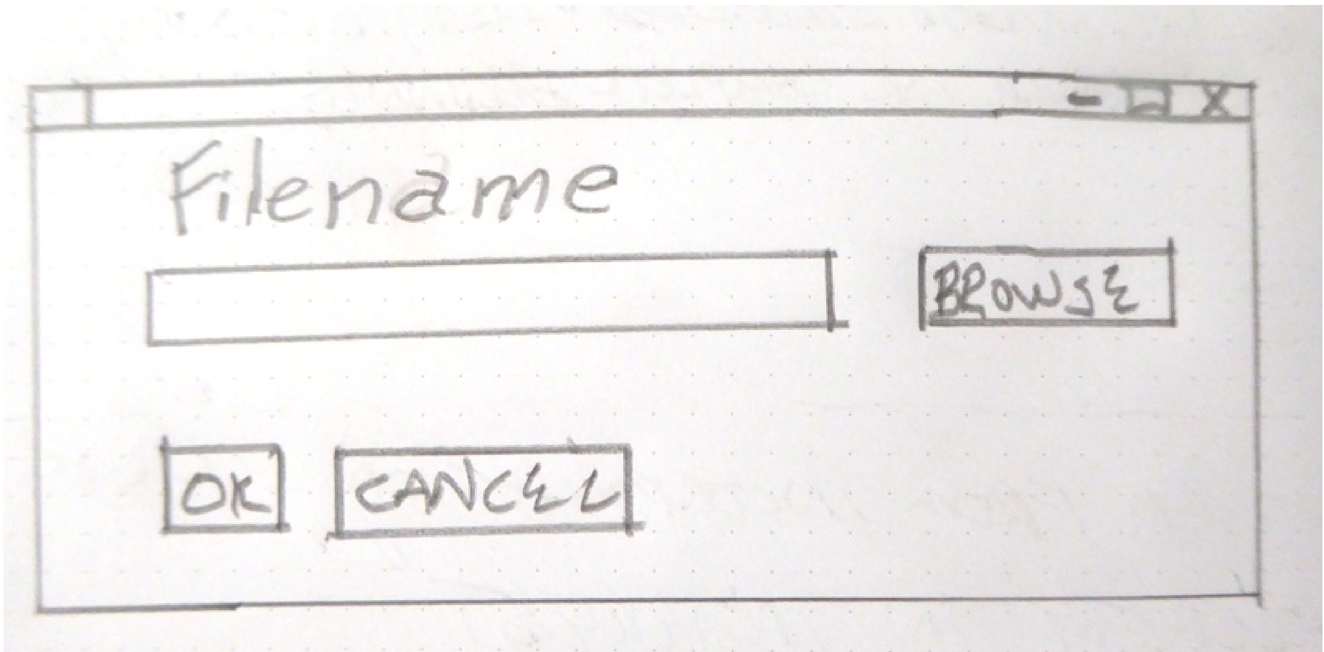
event, values = sg.Window('Enter a number example', layout).Read()

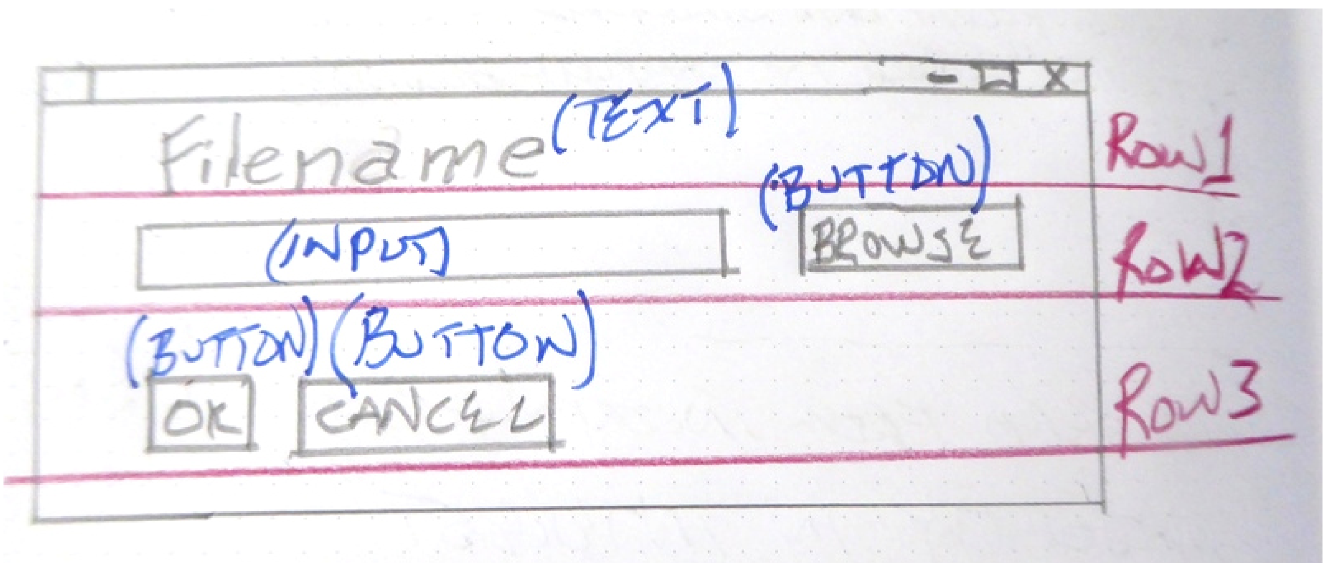
sg.Popup(event, values[0])
```

Your call to `Read` will return a dictionary, but will "look like a list" in how you access it. The first input field will be entry 0, the next one is 1, etc. Later you'll learn about the `key` parameter which allows you to use your own values to identify elements instead of them being numbered for you.

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()] ]

window sg.Window('Get filename example', layout)
event, values = window.read()
window.close()

sg.Popup(event, values[0])
```

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 2 design patterns depending on the type of window you're implementing.

Pattern 1 - "One-shot Window" - Read a window one time then close it

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

The input fields in your window will be returned to you as a dictionary (syntactically it looks just like a list lookup)

```
import PySimpleGUI as sg

layout = [[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', layout)

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

source_filename = values[0]
```

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

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

```
import PySimpleGUI as sg

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

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

while True:
    event, values = window.read()
    if event is None or event == 'Exit':
        break
    print(event, values)

window.close()
```

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

This is a slightly more complex, but maybe more realistic version that reads input from the user and displays that input as text in the window. Your program is likely to be doing both of those activities (input and output) so this will give you a big jump-start.

Do not worry yet what all of these statements mean. Just copy it so you can begin to play with it, make some changes. Experiment to see how thing work.

A final note... the parameter `do_not_clear` in the input call determines the action of the input field after a button event. If this value is True, the input value remains visible following button clicks. If False, then the input field is CLEARED of whatever was input. If you are building a "Form" type of window with data entry, you likely want False. The default is to NOT clear the input element (`do_not_clear=True`).

This example introduces the concept of "keys". Keys are super important in PySimpleGUI as they enable you to identify and work with Elements using names you want to use. Keys can be ANYTHING, except `None` . To access an input element's data that is read in the example below, you will use `values['_IN_']` instead of `values[0]` like before.

```
import PySimpleGUI as sg

layout = [[sg.Text('Your typed chars appear here:'), sg.Text('', key='_OUTPUT_')],
          [sg.Input(key='_IN_')],
          [sg.Button('Show'), sg.Button('Exit')]]

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

while True:
    event, values = window.read()
    print(event, values)
    if event is None or event == 'Exit':
        break
    if event == 'Show':

        window['_OUTPUT_'].update(values['_IN_'])

        window['_OUTPUT_'](values['_IN_'])

window.close()
```

Qt Designer

There actually is a PySimpleGUI Window Designer that uses Qt's window designer. It's outside the scope of this document however. You'll find the project here: https://github.com/nngogol/PySimpleGUIDesigner

I hope to start using it more soon.

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

While one goal was making it simple to create a GUI another just as important goal was to do it in a Pythonic manner. Whether it achieved these goals is debatable, but it was an attempt just the same.

The key to custom windows in PySimpleGUI is to view windows as ROWS of GUI Elements. Each row is specified as a list of these Elements. Put the rows together and you've got a window. This means the GUI is defined as a series of Lists, a Pythonic way of looking at things.

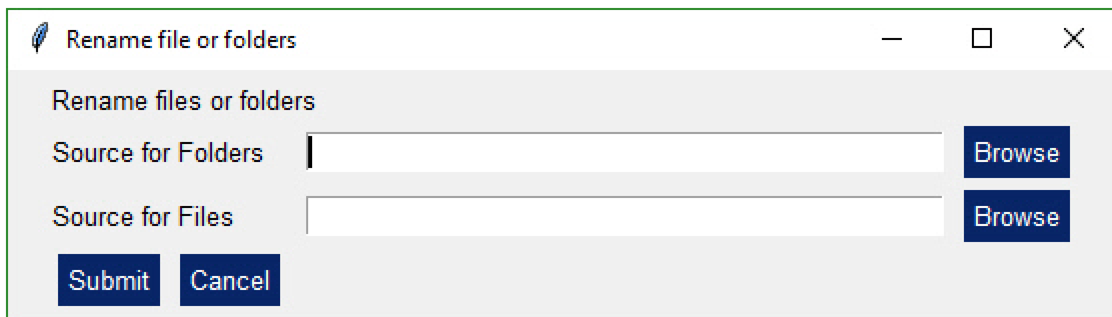
Let's dissect this little program

```
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', layout)

event, values = window.read()
window.close()
folder_path, file_path = values[0], values[1]
print(folder_path, file_path)
```



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 dictionary with 2 values in it. Remember, if you do not specify a `key` when creating an element, one will be created for you. They are ints starting with 0. In this example, we have 2 input elements. They will be addressable as values[0] and values[1]

```
event, values = window.read()
folder_path, file_path = values[0], values[1]
```

In one statement we both show the window and read the user's inputs. In the next line of code the *dictionary* of return values is split into individual variables `folder_path` and `file_path`.

Isn't this what a 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? Most would choose 4.

Return values

There are 2 return values from a call to `Window.read()`, an `event` that caused the `Read` to return and `values` a list or dictionary of values. If there are no elements with keys in the layout, then it will be a list. However, some elements, like some buttons, have a key automatically added to them. **It's best to use keys on all of your input type elements.**

Two Return Values

All Window Read calls return 2 values. By convention a read statement is written:

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

You don't HAVE to write your reads in this way. You can name your variables however you want. But if you want to code them in a way that other programmers using PySimpleGUI are used to, then use this statement.

Events

The first parameter `event` describes **why** the read completed. Events are one of these:

For all Windows:

- Button click
- Window closed using X

For Windows that have specifically enabled these. Please see the appropriate section in this document to learn about how to enable these and what the event return values are.

- Keyboard key press

- Mouse wheel up/down
- Menu item selected
- An Element Changed (slider, spinner, etc)
- A list item was clicked
- Return key was pressed in input element
- Timeout waiting for event
- Text was clicked
- Combobox item chosen
- Table row selected
- etc

Most of the time the event will be a button click or the window was closed. The other Element-specific kinds of events happen when you set `enable_events=True` when you create the Element.

Window closed event

Another convention to follow is the check for windows being closed with an X. *This is an critically important event to catch*. If you don't check for this and you attempt to use the window, your program will crash. Please check for closed window and exit your program gracefully. Your users will like you for it.

Close your windows when you're done with them even though exiting the program will also close them. tkinter can generate an error/warning sometimes if you don't close the window. For other ports, such as PySimpleGUIWeb, not closing the Window will potentially cause your program to continue to run in the background.

To check for a closed window use this line of code:

```
if event is None:
```

Putting it all together we end up with an "event loop" that looks something like this:

```
while True:
   event, values = window.read()
   if event is None:
      break
window.Close()
```

You will very often see the examples and demo programs write this check as:

```
   event, values = window.read()
   if event in (None, 'Exit'):
      break
```

This if statement is the same as:

```
    if event is None or event == 'Exit':
        break
```

Instead of `'Exit'` use the name/key of the button you want to exit the window (Cancel, Quit, etc)

Button Click Events

By default buttons will always return a click event, or in the case of realtime buttons, a button down event. You don't have to do anything to enable button clicks. To disable the events, disable the button using its Update method.

You can enable an additional "Button Modified" event by setting `enable_events=True` in the Button call. These events are triggered when something 'writes' to a button, ***usually*** it's because the button is listed as a "target" in another button.

The button value from a Read call will be one of 2 values: 1. The Button's text - Default 2. The Button's key - If a key is specified

If a button has a key set when it was created, then that key will be returned, regardless of what text is shown on the button. If no key is set, then the button text is returned. If no button was clicked, but the window returned anyway, the event value is the key that caused the event to be generated. For example, if `enable_events` is set on an `Input` Element and someone types a character into that `Input` box, then the event will be the key of the input box.

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:
    event, values = window.read()
    if event is None or event == 'Quit':
        break
```

Actually, the more "Pythonic version" is used in most Demo Programs and examples. They do **exactly** the same thing.

```
while True:
    event, values = window.read()
    if event in (None, 'Quit'):
        break
```

Element Events

Some elements are capable of generating events when something happens to them. For example, when a slider is moved, or list item clicked on or table row clicked on. These events are not enabled by default. To enable events for an Element, set the parameter `enable_events=True`. This is the same as the older `click_submits` parameter. You will find the `click_submits` parameter still in the function definition. You can continue to use it. They are the same setting. An 'or' of the two values is used. In the future, click_submits will be removed so please migrate your code to using `enable_events`.

Name	events
InputText	any change
Combo	item chosen
Listbox	selection changed
Radio	selection changed
Checkbox	selection changed
Spinner	new item selected
Multiline	any change
Text	clicked
Status Bar	clicked
Graph	clicked
Graph	dragged
Graph	drag ended (mouse up)
TabGroup	tab clicked
Slider	slider moved
Table	row selected
Tree	node selected
ButtonMenu	menu item chosen

Name	events
Right click menu	menu item chosen

Other Events

Menubar menu item chosen for MenuBar menus and ButtonMenu menus

You will receive the key for the MenuBar and ButtonMenu. Use that key to read the value in the return values dictionary. The value shown will be the full text plus key for the menu item chosen. Remember that you can put keys onto menu items. You will get the text and the key together as you defined it in the menu definition.

Unlike menu bar and button menus, you will directly receive the menu item text and its key value. You will not do a dictionary lookup to get the value. It is the event code returned from WindowRead().

Windows - keyboard, mouse scroll wheel

Windows are capable of returning keyboard events. These are returned as either a single character or a string if it's a special key. Experiment is all I can say. The mouse scroll wheel events are also strings. Put a print in your code to see what's returned.

Timeouts

If you set a timeout parameter in your read, then the system TIMEOUT_KEY will be returned. If you specified your own timeout key in the Read call then that value will be what's returned instead.

The `values` Variable - Return values as a list

The second parameter from a Read call is either a list or a dictionary of the input fields on the Window.

By default return values are a list of values, one entry for each input field, but for all but the simplest of windows the return values will be a dictionary. This is because you are likely to use a 'key' in your layout. When you do, it forces the return values to be a dictionary.

Each of the Elements that are Input Elements will have a value in the list of return values. If you know for sure that the values will be returned as a list, then you could get clever and unpack directly into variables.

event, (filename, folder1, folder2, should_overwrite) = sg.Window('My title', window_rows).Read()

Or, more commonly, you can unpack the return results separately. This is the preferred method because it works for **both** list and dictionary return values.

```
event, values = sg.Window('My title', window_rows).Read()
event, value_list = window.read()
value1 = value_list[0]
value2 = value_list[1]
    ...
```

However, this method isn't good when you have a lot of input fields. If you insert a new element into your window then you will have to shuffle your unpacks down, modifying each of the statements to reference `value_list[x]`.

The more common method is to request your values be returned as a dictionary by placing keys on the "important" elements (those that you wish to get values from and want to interact with)

`values` Variable - Return values as a dictionary

For those of you that have not encountered a Python dictionary, don't freak out! Just copy and paste the sample code and modify it. Follow this design pattern and you'll be fine. And you might learn something along the way.

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:

`window = sg.Window("My title", layout).Read()`

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

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()]
          ]

window = sg.Window('Simple data entry window', layout)
event, values = window.read()
window.Close()

sg.Popup(event, 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_']
```

Think of the variable values in the same way as you would a list, however, instead of using 0,1,2, to reference each item in the list, use the values of the key. The Name field in the window above is referenced by `values['_NAME_']`.

You will find the key field used quite heavily in most PySimpleGUI windows unless the window is very simple.

One convention you'll see in many of the demo programs is keys being named in all caps with an underscores at the beginning and the end. You don't HAVE to do this... your key value may look like this: `key = '_NAME__'`

The reason for this naming convention is that when you are scanning the code, these key values jump out at you. You instantly know it's a key. Try scanning the code above and see if those keys pop out. `key = '_NAME__'`

The Event Loop / Callback Functions

All GUIs have one thing in common, an "event loop". Usually the GUI framework runs the event loop for you, but sometimes you want greater control and will run your own event loop. You often hear the term event loop when discussing embedded systems or on a Raspberry Pi.

With PySimpleGUI if your window will remain open following button clicks, then your code will have an event loop. If your program shows a single "one-shot" window, collects the data and then has no other GUI interaction, then you don't need an event loop.

There's nothing mysterious about event loops... they are loops where you take care of.... wait for it..... *events*. Events are things like button clicks, key strokes, mouse scroll-wheel up/down.

This little program has a typical PySimpleGUI Event Loop.

The anatomy of a PySimpleGUI event loop is as follows, *generally speaking*. * The actual "loop" part is a `while True` loop * "Read" the event and any input values the window has * Check to see if window was closed or user wishes to exit * A series of `if event ....` statements

Here is a complete, short program to demonstrate each of these concepts.

```
import PySimpleGUI as sg

sg.ChangeLookAndFeel('GreenTan')


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


column1 = [[sg.Text('Column 1', background_color='lightblue', 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('(Almost) All 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, tick_interval=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='lightblue')]])],
    [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 form'), sg.Cancel()]]

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

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

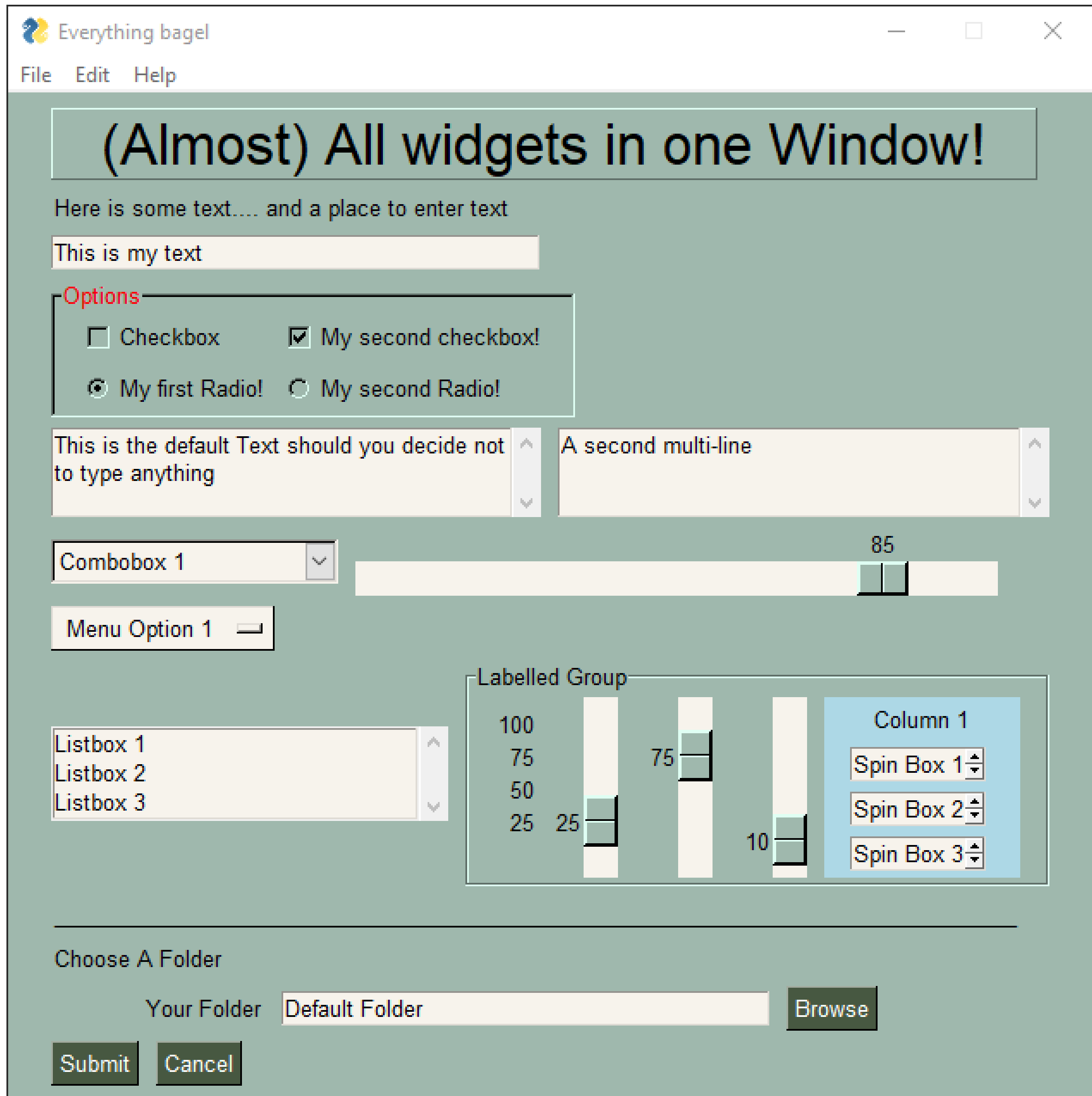
This is a complex window with quite a bit of custom sizing to make things line up well. This is code you only have to write once. When looking at the code, remember that what you're seeing is a list of lists. Each row contains a list of Graphical Elements that are used to create the window. If you see a pair of square brackets [ ] then you know you're reading one of the rows. Each row of your GUI will be one of these lists.

This window may look "ugly" to you which is because no effort has been made to make it look nice. It's purely functional. There are 30 Elements in the window. THIRTY Elements. Considering what it does, it's miraculous or in the least incredibly impressive. Why? Because in less than 50 lines of code that window was created, shown, collected the results and showed the results in another window.

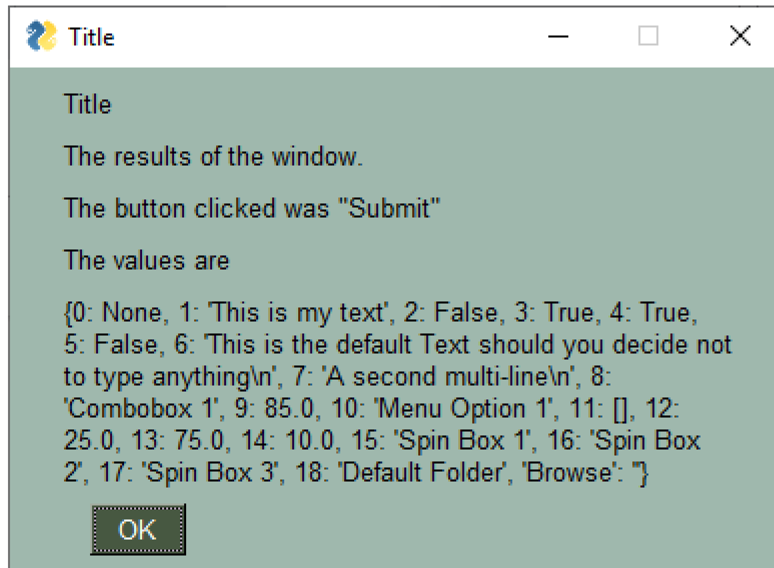
50 lines. It'll take you 50 lines of tkinter or Qt code to get the first 3 elements of the window written, if you can even do that.

No, let's be clear here... this window will take a massive amount of code using the conventional Python GUI packages. It's a fact and if you care to prove me wrong, then by ALL means PLEASE do it. Please write this window using tkinter, Qt, or WxPython and send the code!

Note this window even has a menubar across the top, something easy to miss.



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



`Note, event values can be None`. The value for `event` will be the text that is displayed on the button element when it was created or the key for the button. If the user closed the window using the "X" in the upper right corner of the window, then `event` will be `None`. It is ***vitally important*** that your code contain the proper checks for None.

For "persistent windows", **always give your users a way out of the window**. Otherwise you'll end up with windows that never properly close. It's literally 2 lines of code that you'll find in every Demo Program. While you're at it, make sure a `window.Close()` call is after your event loop so that your window closes for sure.

You can see in the results Popup window that the values returned are a dictionary. Each input field in the window generates one item in the return values list. Input fields often return a `string`. Check Boxes and Radio Buttons return `bool`. Sliders return float or perhaps int depending on how you configured it or which port you're using.

If your window has no keys and it has no buttons that are "browse" type of buttons, then it will return values to you as a list instead of a dictionary. If possible PySimpleGUI tries to return the values as a list to keep things simple.

Note in the list of return values in this example, many of the keys are numbers. That's because no keys were specified on any of the elements (although one was automatically made for you). If you don't specify a key for your element, then a number will be sequentially assigned. For elements that you don't plan on modifying or reading values from, like a Text Element, you can skip adding keys. For other elements, you'll likely want to add keys so that you can easily access the values and perform operations on them.

Operations That Take a "Long Time"

If you're a Windows user you've seen windows show in their title bar "Not Responding" which is soon followed by a Windows popop stating that "Your program has stopped responding". Well, you too can make that message and popup appear if you so wish! All you need to do is execute an operation that takes "too long" (i.e. a few seconds) inside your event loop.

You have a couple of options for dealing this with. If your operation can be broken up into smaller parts, then you can call `Window.Refresh()` occassionally to avoid this message. If you're running a loop for example, drop that call in with your other work. This will keep the GUI happy and Window's won't complain.

If, on the other hand, your operation is not under your control or you are unable to add `Refresh` calls, then the next option available to you is to move your long operations into a thread.

There are a couple of demo programs available for you to see how to do this. You basically put your work into a thread. When the thread is completed, it tells the GUI by sending a message through a queue. The event loop will run with a timer set to a value that represents how "responsive" you want your GUI to be to the work completing.

These 2 demo programs are called

```
Demo_Threaded_Work.py - Best documented.  Single thread used for long task
Demo_Multithreaded_Long_Tasks.py - Similar to above, but with less fancy GUI. Allows you to set amount of time
```

These 2 particular demos have a LOT of comments showing you where to add your code, etc. The amount of code to do this is actually quite small and you don't need to understand the mechanisms used if you simply follow the demo that's been prepared for you.

Multitheaded Programs

While on the topic of multiple threads, another demo was prepared that shows how you can run multiple threads in your program that all communicate with the event loop in order to display something in the GUI window. Recall that for PySimpleGUI (at least the tkinter port) you cannot make PySimpleGUI calls in threads other than the main program thread.

The key to these threaded programs is communication from the threads to your event loop. The mechanism chosen for these demonstrations uses the Python built-in `queue` module. The event loop polls these queues to see if something has been sent over from one of the threads to be displayed.

You'll find the demo that shows multiple threads communicating with a single GUI is called:

```
Demo_Multithreaded_Queued.py
```

Once again a **warning** is in order for plain PySimpleGUI (tkinter based) - your GUI must never run as anything but the main program thread and no threads can directly call PySimpleGUI calls.

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 / Asynchronous 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. You'll know it blocks if the `Read` call has no timeout parameter.

A blocking Read (one that waits until an event happens) look like this:

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

A non-blocking / Async Read call looks like this:

```
event, values = window.Read(timeout=100)
```

You can learn more about these async / non-blocking windows toward the end of this document.

Window Object - Beginning a window

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

Note - There is no direct support for "**modal windows**" in PySimpleGUI. All windows are accessable at all times unless you manually change the windows' settings.

IMPORTANT - Many of the `Window` methods require you to either call `Window.Read` or `Window.Finalize` (or set `finalize=True` in your `Window` call) before you call the method. This is because these 2 calls are what actually creates the window using the underlying GUI

Framework. Prior to one of those calls, the methods are likely to crash as they will not yet have their underlying widgets created.

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.

Multiple Monitors and Linux

The auto-centering (default) location for your PySimpleGUI window may not be correct if you have multiple monitors on a Linux system. On Windows multiple monitors appear to work ok as the primary monitor the tkinter utilizes and reports on.

Linux users with multiple monitors that have a problem when running with the default location will need to specify the location the window should be placed when creating the window by setting the `location` parameter.

Window Size

You can get your window's size by access the `Size` property. The window has to be Read once or Finalized in order for the value to be correct. Note that it's a property, not a call.

`my_windows_size = window.Size`

To finalize your window:

```
window = Window('My Title', layout).Finalize()
```

If using PySimpleGUI 4.2 and later:

```
window = Window('My Title', layout, finalize=True)
```

Element Sizes

There are multiple ways to set the size of Elements. They are:

1. The global default size - change using `SetOptions` function
2. At the Window level - change using the parameter `default_element_size` in your call to `Window`
3. At the Element level - each element has a `size` parameter

Element sizes are measured in characters (there are exceptions). A Text Element with `size = (20,1)` has a size of 20 characters wide by 1 character tall.

The default Element size for PySimpleGUI is `(45,1)`.

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

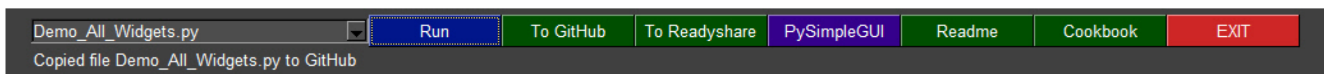
Should you wish to create cool looking windows that are clean with no windows titlebar, use the no_titlebar option when creating the window.

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.

Linux users! Note that this setting has side effects for some of the other Elements. Multi-line input doesn't work at all, for example So, use with caution.



Grab Anywhere

This is a feature unique to PySimpleGUI.

Note - 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.

Focus

PySimpleGUI will set a default focus location for you. This generally means the first input field. You can set the focus to a particular element. If you are going to set the focus yourself, then you should turn off the automatic focus by setting `use_default_focus=False` in your Window call.

Closing Windows

When you are completely done with a window, you should close it and then delete it so that the resources, in particular the tkinter resources, are properly cleaned up.

If you wish to do this in 1 line of code, here's your line:

```
window.close(); del window
```

The delete helps with a problem multi-threaded application encounter where tkinter complains that it is being called from the wrong thread (not the program's main thread)

Window Methods That Complete Formation of Window

After you have completed making your layout, stored in a variable called `layout` in these examples, you will create your window.

The creation part of a window involves 3 steps.

1. Create a `Window` object
2. Adding your Layout to the window
3. Optional - Finalize if want to make changes prior to `Read` call

Over time the PySimpleGUI code has continued to compact, compress, so that as little code as possible will need to be written by the programmer.

The Individual Calls

This is the "long form" as each method is called individually.

```
window = sg.Window('My Title')
window.Layout(layout)
window.Finalize()
```

Chaining The Calls (the old method)

The next level of compression that was done was to chain the calls together into a single line of code.

```
window = sg.Window('My Title').Layout(layout).Finalize()
```

Using Parameters Instead of Calls (New Preferred Method)

Here's a novel concept, instead of using chaining, something that's foreign to beginners, use parameters to the `Window` call. And that is exactly what's happened as of 4.2 of the PySimpleGUI port.

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

Rather than pushing the work onto the user of doing the layout and finalization calls, let the Window initialization code do it for you. Yea, it sounds totally obvious now, but it didn't a few months ago.

This capability has been added to all 4 PySimpleGUI ports but none are on PyPI just yet as there is some runtime required first to make sure nothing truly bad is going to happen.

Call to set the window layout. Must be called prior to `Read`. Most likely "chained" in line with the Window creation.

```
window = sg.Window('My window title', layout)
```

`Finalize()` or `Window` parameter `finalize=True`

Call to force a window to go through the final stages of initialization. This will cause the tkinter resources to be allocated so that they can then be modified. This also causes your window to appear. If you do not want your window to appear when Finalize is called, then set the Alpha to 0 in your window's creation parameters.

If you want to call an element's `Update` method or call a `Graph` element's drawing primitives, you ***must*** either call `Read` or `Finalize` prior to making those calls.

Read(timeout=None, timeout_key=TIMEOUT_KEY)

Read the Window's input values and button clicks in a blocking-fashion

Returns event, values. Adding a timeout can be achieved by setting timeout=*number of milliseconds* before the Read times out after which a "timeout event" is returned. The value of timeout_key will be returned as the event. If you do not specify a timeout key, then the value `TIMEOUT_KEY` will be returned.

If you set the timeout = 0, then the Read will immediately return rather than waiting for input or for a timeout. It's a truly non-blocking "read" of the window.

Layouts

While at this point in the documentation you've not been shown very much about each Element available, you should read this section carefully as you can use the techniques you learn in here to build better, shorter, and easier to understand PySimpleGUI code.

If it feels like this layout section is too much too soon, then come back to this section after you're learned about each Element. **Whatever order you find the least confusing is the best.**

While you've not learned about Elements yet, it makes sense for this section to be up front so that you'll have learned how to use the elements prior to learning how each element works. At this point in your PySimpleGUI education, it is better for you to grasp time efficient ways of working with Elements than what each Element does. By learning now how to assemble Elements now, you'll have a good model to put the elements you learn into.

There are *several* aspects of PySimpleGUI that make it more "Pythonic" than other Python GUI SDKs. One of the areas that is unique to PySimpleGUI is how a window's "layout" is defined, specified or built. A window's "layout" is simply a list of lists of elements. As you've already learned, these lists combine to form a complete window. This method of defining a window is super-powerful because lists are core to the Python language as a whole and thus are very easy to create and manupulate.

Think about that for a moment and compare/contrast with Qt, tkinter, etc. With PySimpleGUI the location of your element in a matrix determines where that Element is shown in the window. It's so ***simple*** and that makes it incredibly powerful. Want to switch a row in your GUI that has text with the one below it that has an input element? No problem, swap the lines of code and you're done.

Layouts were designed to be visual. The idea is for you to be able to envision how a window will look by simplyh looking at the layout in the code. The CODE itself matches what is drawn on the screen. PySimpleGUI is a cross between straight Python code and a visual GUI designer.

In the process of creating your window, you can manipulate these lists of elements without having an impact on the elements or on your window. Until you perform a "layout" of the list, they are nothing more than lists containing objects (they just happen to be your window's elements).

Many times your window definition / layout will be a static, straightforward to create.

However, window layouts are not limited to being one of these staticly defined list of Elements.

Generated Layouts (For sure want to read if you have > 5 repeating elements/rows)

There are 5 specific techniques of generating layouts discussed in this section. They can be used alone or in combination with each other.

1. Layout + Layout concatenation `[[A]] + [[B]] = [[A], [B]]`
2. Element Addition on Same Row `[[A] + [B]] = [[A, B]]`
3. List Comprehension to generate a row `[A for x in range(10)] = [A,A,A,A,A...]`
4. List Comprehension to generate multiple rows `[[A] for x in range(10)] = [[A],[A],...]`
5. User Defined Elements / Comound Elements

Example - List Comprehension To Concatenate Multiple Rows - "To Do" List Example

Let's create a little layout that will be used to make a to-do list using PySimpleGUI.

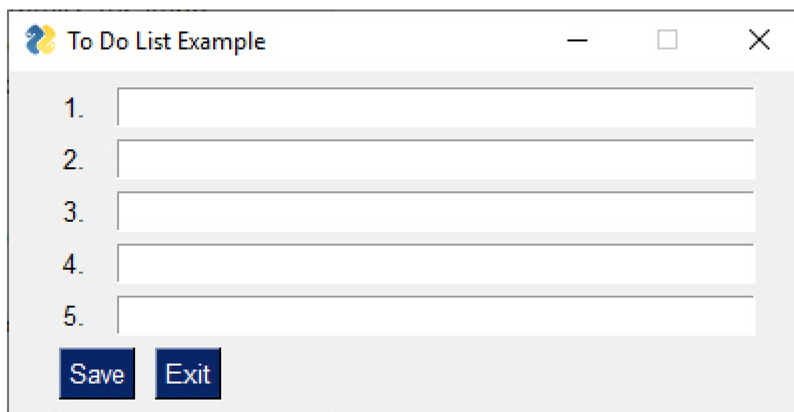
Brute Force

```
import PySimpleGUI as sg

layout = [
            [sg.Text('1. '), sg.In(key=1)],
            [sg.Text('2. '), sg.In(key=2)],
            [sg.Text('3. '), sg.In(key=3)],
            [sg.Text('4. '), sg.In(key=4)],
            [sg.Text('5. '), sg.In(key=5)],
            [sg.Button('Save'), sg.Button('Exit')]
         ]

window = sg.Window('To Do List Example', layout)
event, values = window.read()
```

The output from this script was this window:



Take a moment and look at the code and the window that's generated. Are you able to look at the layout and envision the Window on the screen?

Build By Concatenating Rows

The brute force method works great on a list that's 5 items long, but what if your todo list had 40 items on it. THEN what? Well, that's when we turn to a "generated" layout, a layout that is generated by your code. Replace the layout= stuff from the previous example with this definition of the layout.

```
import PySimpleGUI as sg

layout = []
for i in range(1,6):
    layout += [sg.Text(f'{i}. '), sg.In(key=i)],
layout += [[sg.Button('Save'), sg.Button('Exit')]]

window = sg.Window('To Do List Example', layout)
event, values = window.read()
```

It produces the exact same window of course. That's progress.... went from writing out every row of the GUI to generating every row. If we want 48 items as suggested, change the range(1,6) to range(1,48). Each time through the list another row is added onto the layout.

Create Several Rows Using List Comprehension

BUT, we're not done yet!

This is **Python**, we're using lists to build something up, so we should be looking at ***list comprehensions***. Let's change the `for` loop into a list comprehension. Recall that our `for` loop was used to concatenate 6 rows into a layout.

```
layout =  [[sg.Text(f'{i}. '), sg.In(key=i)] for i in range(1,6)]
```

Here we've moved the `for` loop to inside of the list definition (a list comprehension)

Concatenating Multiple Rows

We have our rows built using the list comprehension, now we just need the buttons. They can be easily "tacked onto the end" by simple addition.

```
layout =  [[sg.Text(f'{i}. '), sg.In(key=i)] for i in range(1,6)]
layout += [[sg.Button('Save'), sg.Button('Exit')]]
```

Anytime you have 2 layouts, you can concatenate them by simple addition. Make sure your layout is a "list of lists" layout. In the above example, we know the first line is a generated layout of the input rows. The last line adds onto the layout another layout... note the format being [[ ]].

This button definition is an entire layout, making it possible to add to our list comprehension

`[[sg.Button('Save'), sg.Button('Exit')]]`

It's quite readable code. The 2 layouts line up visually quite well.

But let's not stop there with compressing the code. How about removing that += and instead change the layout into a single line with just a `+` between the two sets of row.

Doing this concatenation on one line, we end up with this single statement that creates the **entire layout** for the GUI:

```
layout =  [[sg.Text(f'{i}. '), sg.In(key=i)] for i in range(1,6)] + [[sg.Button('Save'), sg.Button('Exit')]]
```

Final "To Do List" Program

And here we have our final program... all **4** lines.

```
import PySimpleGUI as sg

layout  = [[sg.Text(f'{i}. '), sg.In(key=i)] for i in range(1,6)] + [[sg.Button('Save'), sg.Button('Exit')]]

window = sg.Window('To Do List Example', layout)

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

If you really wanted to crunch things down, you can make it a 2 line program (an import and 1 line of code) by moving the layout into the call to `Window`

```
import PySimpleGUI as sg

event, values = sg.Window('To Do List Example', layout=[[sg.Text(f'{i}. '), sg.In(key=i)] for i in range(1,6)] +
[[sg.Button('Save'), sg.Button('Exit')]]).Read()
```

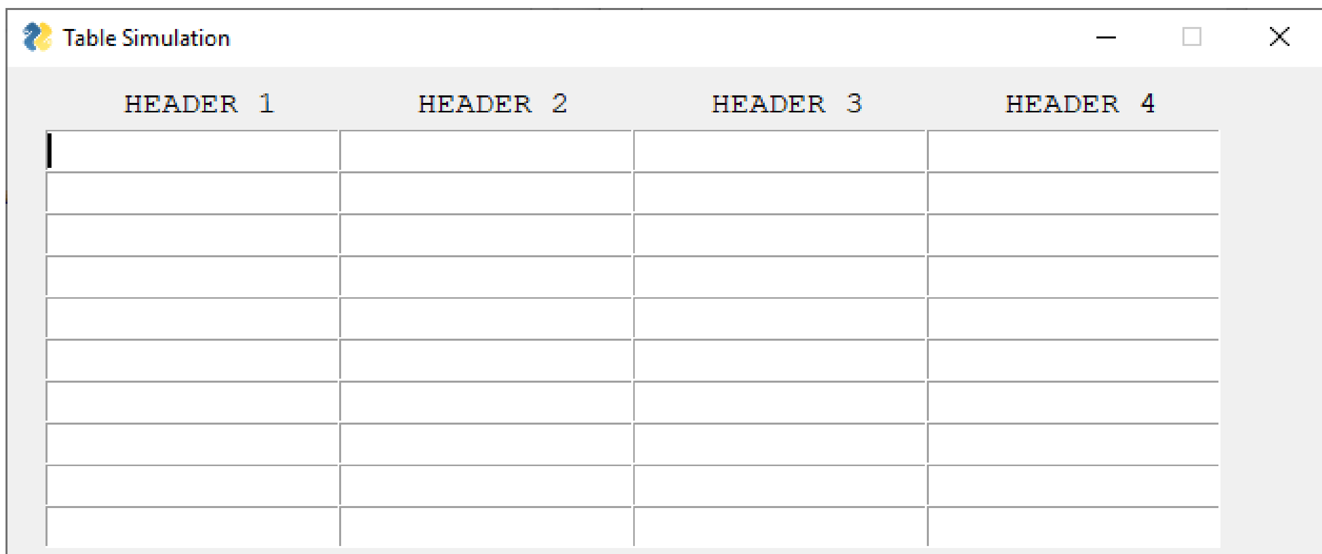
Example - List Comprehension to Build Rows - Table Simulation - Grid of Inputs

In this example we're building a "table" that is 4 wide by 10 high using `Input` elements

The end results we're seeking is something like this:

HEADER 1	HEADER 2	HEADER 3	HEADER 4
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT
INPUT	INPUT	INPUT	INPUT

Once the code is completed, here is how the result will appear:



We're going to be building each row using a list comprehension and we'll build the table by concatenating rows using another list comprehension. That's a list comprehension that goes across and another list comprehension that goes down the layout, adding one row after another.

Building the Header

First let's build the header. There are 2 concepts to notice here:

```
import PySimpleGUI as sg

headings = ['HEADER 1', 'HEADER 2', 'HEADER 3','HEADER 4']
header =  [[sg.Text('  ')] + [sg.Text(h, size=(14,1)) for h in headings]]
```

There are 2 things in this code to note 1. The list comprehension that makes the heading elements 2. The spaces added onto the front

Let's start with the headers themselves.

This is the code that makes a row of Text Elements containing the text for the headers. The result is a list of Text Elements, a row.

```
[sg.Text(h, size=(14,1)) for h in headings]
```

Then we add on a few spaces to shift the headers over so they are centered over their columns. We do this by simply adding a `Text` Element onto the front of that list of headings.

```
header =  [[sg.Text('  ')] + [sg.Text(h, size=(14,1)) for h in headings]]
```

This `header` variable is a layout with 1 row that has a bunch of `Text` elements with the headings.

Building the Input Elements

The `Input` elements are arranged in a grid. To do this we will be using a double list comprehension. One will build the row the other will add the rows together to make the grid. Here's the line of code that does that:

```
input_rows = [[sg.Input(size=(15,1), pad=(0,0)) for col in range(4)] for row in range(10)]
```

This portion of the statement makes a single row of 4 `Input` Elements

```
[sg.Input(size=(15,1), pad=(0,0)) for col in range(4)]
```

Next we take that list of `Input` Elements and make as many of them as there are rows, 10 in this case. We're again using Python's awesome list comprehensions to add these rows together.

```
input_rows = [[sg.Input(size=(15,1), pad=(0,0)) for col in range(4)] for row in range(10)]
```

The first part should look familiar since it was just discussed as being what builds a single row. To make the matrix, we simply take that single row and create 10 of them, each being a list.

Putting it all together

Here is our final program that uses simple addition to add the headers onto the top of the input matrix.

```
import PySimpleGUI as sg

headings = ['HEADER 1', 'HEADER 2', 'HEADER 3','HEADER 4']
header =  [[sg.Text('  ')] + [sg.Text(h, size=(14,1)) for h in headings]]

input_rows = [[sg.Input(size=(15,1), pad=(0,0)) for col in range(4)] for row in range(10)]

layout = header + input_rows

window = sg.Window('Table Simulation', layout, font='Courier 12')
event, values = window.read()
```

User Defined Elements / Compound Elements

"User Defined Elements" and "Compound Elements" are one or more PySimpleGUI Elements that are wrapped in a function definition. In a layout, they have the appearance of being a custom elements of some type.

User Defined Elements are particularly useful when you set a lot of parameters on an element that you use over and over in your layout.

Example - A Grid of Buttons for Calculator App

Let's say you're making a calculator application with buttons that have these settings:

- font = Helvetica 20
- size = 5,1
- button color = white on blue

The code for **one** of these buttons is:

```
sg.Button('1', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20))
```

If you have 6 buttons across and 5 down, your layout will have 30 of these lines of text.

One row of these buttons could be written:

```
    [sg.Button('1', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20)),
     sg.Button('2', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20)),
     sg.Button('3', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20)),
     sg.Button('log', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20)),
     sg.Button('ln', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20)),
     sg.Button('-', button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20))],
```

By using User Defined Elements, you can significantly shorten your layouts. Let's call our element `CBtn`. It would be written like this:

```
def CBtn(button_text):
    return sg.Button(button_text, button_color=('white', 'blue'), size=(5, 1), font=("Helvetica", 20))
```

Using your new `CBtn` Element, you could rewrite the row of buttons above as:

```
[CBtn('1'), CBtn('2'), CBtn('3'), CBtn('log'), CBtn('ln'), CBtn('-')],
```

See the tremendous amount of code you do not havew to write! USE this construct any time you find yourself copying an element many times.

But let's not stop there.

Since we've been discussing list comprehensions, let's use them to create this row. The way to do that is to make a list of the symbols that go across the row make a loop that steps through that list. The result is a list that looks like this:

```
[CBtn(t) for t in ('1','2','3', 'log', 'ln', '-')],
```

That code produces the same list as this one we created by hand:

```
[CBtn('1'), CBtn('2'), CBtn('3'), CBtn('log'), CBtn('ln'), CBtn('-')],
```

Compound Elements

Just like a `Button` can be returned from a User Defined Element, so can multiple Elements.

Going back to the To-Do List example we did earlier, we could have defined a User Defined Element that represented a To-Do Item and this time we're adding a checkbox. A single line from this list will be:

- The item # (a `Text` Element)
- A `Checkbox` Element to indicate completed
- An `Input` Element to type in what to do

The definition of our User Element is this `ToDoItem` function. It is a single User Element that is a combination of 3 PySimpleGUI Elements.

```
def ToDoItem(num):
    return [sg.Text(f'{num}. '), sg.CBox(''), sg.In()]
```

This makes creating a list of 5 to-do items downright trivial when combined with the list comprehension techniques we learned earlier. Here is the code required to create 5 entries in our to-do list.

```
layout = [ToDoItem(x) for x in range(1,6)]
```

We can then literally add on the buttons

```
layout = [ToDoItem(x) for x in range(1,6)] + [[sg.Button('Save'), sg.Button('Exit')]]
```

And here is our final program

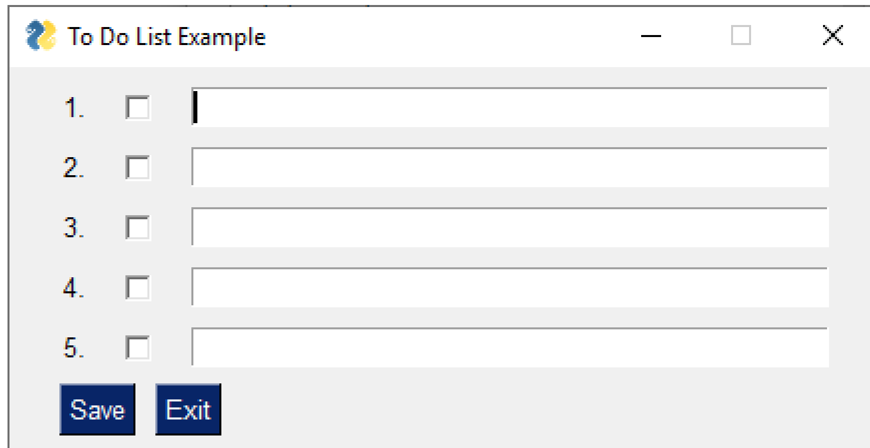
```
import PySimpleGUI as sg

def ToDoItem(num):
    return [sg.Text(f'{num}. '), sg.CBox(''), sg.In()]

layout = [ToDoItem(x) for x in range(1,6)] + [[sg.Button('Save'), sg.Button('Exit')]]

window = sg.Window('To Do List Example', layout)
event, values = window.read()
```

And the window it creates looks like this:



Elements

You will find information on Elements and all other classes and functions are located near the end of this manual. They are in 1 large section of the readme, in alphabetical order for easy lookups. This section's discussion of Elements is meant to teach you how they work. The other section has detailed call signatures and parameter definitions.

"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 ("Button" & all shortcut buttons)
    - Realtime
- Checkboxes

- Radio Buttons
- Listbox
- Slider
- Multi-line Text Input/Output
- Multi-line Text Output (not on tkinter version)
- Scroll-able Output
- Vertical Separator
- Progress Bar
- Option Menu
- Menu
- ButtonMenu
- Frame
- Column
- Graph
- Image
- Table
- Tree
- Tab, TabGroup
- StatusBar
- Pane
- Stretch (Qt only)
- Sizer (plain PySimpleGUI only)

Common Element Parameters

Some parameters that you will see on almost all Element creation calls include:

- key - Used with window.FindElement and with return values
- tooltip - Hover your mouse over the elemnt and you'll get a popup with this text
- size - (width, height) - usually measured in characters-wide, rows-high. Sometimes they mean pixels
- font - specifies the font family, size, etc
- colors - Color name or #RRGGBB string
- pad - Amount of padding to put around element
- enable_events - Turns on the element specific events
- visible - Make elements appear and disappear

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. You can change the color of the background of the tooltip on the tkinter version of PySimpleGUI by setting `TOOLTIP_BACKGROUND_COLOR` to the color string of your choice. The default value for the color is:

`TOOLTIP_BACKGROUND_COLOR = "#ffffe0"`

Size

Info on setting default element sizes is discussed in the Window section above.

Specifies the amount of room reserved for the Element. For elements that are character based, such a Text, it is (# characters, # rows). Sometimes it is a pixel measurement such as the Image element. And sometimes a mix like on the Slider element (characters long by pixels wide).

Some elements, Text and Button, have an auto-size setting that is `on` by default. It will size the element based on the contents. The result is that buttons and text fields will be the size of the string creating them. You can turn it off. For example, for Buttons, the effect will be that all buttons will be the same size in that window.

Element Sizes - Non-tkinter Ports (Qt, WxPython, Web)

In non-tkinter ports you can set the specific element sizes in 2 ways. One is to use the normal `size` parameter like you're used to using. This will be in characters and rows.

The other way is to use a new parameter, `size_px`. This parameter allows you to specify the size directly in pixels. A setting of `size_px=(300,200)` will create an Element that is 300 x 200 pixels.

Additionally, you can also indicate pixels using the `size` parameter, **if the size exceeds the threshold for conversion.** What does that mean? It means if your width is > 20 (`DEFAULT_PIXEL_TO_CHARS_CUTOFF`), then it is assumed you're talking pixels, not characters. However, some of the "normally large" Elements have a cutoff value of 100. These include, for example, the `Multline` and `Output` elements.

If you're curious about the math used to do the character to pixels conversion, it's quite crude, but functional. The conversion is completed with the help of this variable:

`DEFAULT_PIXELS_TO_CHARS_SCALING = (10,26)`

The conversion simply takes your `size[0]` and multiplies by 10 and your `size[1]` and multiplies it by 26.

Colors

A string representing color. Anytime colors are involved, you can specify the tkinter color name such as 'lightblue' or an RGB hex value '#RRGGBB'. For buttons, the color parameter is a tuple (text color, background color)

Anytime colors are written as a tuple in PySimpleGUI, the way to figure out which color is the background is to replace the "," with the word "on". ('white', 'red') specifies a button that is "white on red". Works anywhere there's a color tuple.

Pad

The amount of room around the element in pixels. The default value is (5,3) which means leave 5 pixels on each side of the x-axis and 3 pixels on each side of the y-axis. You can change this on a global basis using a call to SetOptions, or on an element basis.

If you want more pixels on one side than the other, then you can split the number into 2 number. If you want 200 pixels on the left side, and 3 pixels on the right, the pad would be ((200,3), 3). In this example, only the x-axis is split.

Font

Specifies the font family, size, and style. Font families on Windows include: * Arial * Courier * Comic, * Fixedsys * Times * Verdana * Helvetica (the default I think)

The fonts will vary from system to system, however, Tk 8.0 automatically maps Courier, Helvetica and Times to their corresponding native family names on all platforms. Also, font families cannot cause a font specification to fail on Tk 8.0 and greater.

If you wish to leave the font family set to the default, you can put anything not a font name as the family. The PySimpleGUI Demo programs and documentation use the family 'Any' to demonstrate this fact.. You could use "default" if that's more clear to you.

There are 2 formats that can be used to specify a font... a string, and a tuple Tuple - (family, size, styles) String - "Family Size Styles"

To specify an underlined, Helvetica font with a size of 15 the values: ('Helvetica', 15, 'underline italics') 'Helvetica 15 underline italics'

Key

If you are going to do anything beyond the basic stuff with your GUI, then you need to understand keys. Keys are a way for you to "tag" an Element with a value that will be used to identify that element. After you put a key in an element's definition, the values returned from Read will use that key to tell you the value. For example, if you have an input field:

`Input(key='mykey')`

And your read looks like this: `event, values = Read()`

Then to get the input value from the read it would be: `values['mykey']`

You also use the same key if you want to call Update on an element. Please see the section below on Updates to understand that usage.

Keys can be ANYTHING. Let's say you have a window with a grid of input elements. You could use their row and column location as a key (a tuple)

`key=(row, col)`

Then when you read the `values` variable that's returned to you from calling `Window.Read()`, the key in the `values` variable will be whatever you used to create the element. In this case you would read the values as: `values[(row, col)]`

Most of the time they are simple text strings. In the Demo Programs, keys are written with this convention: `_KEY_NAME_` (underscore at beginning and end with all caps letters) or '-KEY_NAME-. You don't have to follow that convention. It's used so that you can quickly spot when a key is being used.

To find an element's key, access the member variable `.Key` for the element. This assumes you've got the element in a variable already.

text_elem = sg.Text('', key='-TEXT-')

the_key = text_elem.Key

Visible

Beginning in version 3.17 you can create Elements that are initially invisible that you can later make visible.

To create an invisible Element, place the element in the layout like you normally would and add the parameter

`visible=False`.

Later when you want to make that Element visible you simply call the Element's `Update` method and pass in the parameter `visible=True`

This feature works best on Qt, but does work on the tkinter version as well. The visible parameter can also be used with the Column and Frame "container" Elements.

Note - Tkiner elements behave differently than Qt elements in how they arrange themselves when going from invisible to visible.

Tkinet elements tend to STACK themselves.

One workaround is to place the element in a Column with other elements on its row. This will hold the place of the row it is to be placed on. It will move the element to the end of the row however.

If you want to not only make the element invisible, on tkinter you can call `Element.

Qt elements tend to hold their place really well and the window resizes itself nicely. It is more precise and less klunky.

Shortcut Functions / Multiple Function Names

Perhaps not the best idea, but one that's done none the less is the naming of methods and functions. Some of the more "Heavily Travelled Elements" (and methods/functions) have "shortcuts".

In other words, I am lazy and don't like to type. The result is multiple ways to do exactly the same thing. Typically, the Demo Programs and other examples use the full name, or at least a longer name. Thankfully PyCharm will show you the same documentation regardless which you use.

This enables you to code much quicker once you are used to using the SDK. The Text Element, for example, has 3 different names `Text` , `Txt` or `T` . InputText can also be written `Input` or `In` .

The shortcuts aren't limited to Elements. The `Window` method `Window.FindElement` can be written as `Window.Element` because it's such a commonly used function. BUT,even that has now been shortened.

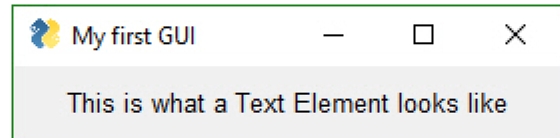
It's an ongoing thing. If you don't stay up to date and one of the newer shortcuts is used, you'll need to simply rename that shortcut in the code. For examples Replace sg.T with sg.Text if your version doesn't have sg.T in it.

Text Element | `T == Txt == Text`

Basic Element. It displays text.

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

When creating a Text Element that you will later update, make sure you reserve enough characters for the new text. When a Text Element is created without a size parameter, it is created to exactly fit the characters provided.



With proportional spaced fonts (normally the default) the pixel size of one set of characters will differ from the pixel size of a different set of characters even though the set is of the same number of characters. In other words, not all letters use the same number of pixels. Look at the text you're reading right now and you will see this. An "i" takes up a less space then an "A".

`Window.FindElement(key)` Shortcut `Window[key]`

There's been a fantastic leap forward in making PySimpleGUI code more compact.

Instead of writing:

window.FindElement(key).Update(new_value)
```

You can now write it as:

```python
window[key].Update(new_value)
```

This change has been released to PyPI for PySimpleGUI

MANY Thanks is owed to the person that suggested and showed me how to do this. It's an incredible find.

`Element.Update()` -> `Element()` shortcut

This has to be one of the strangest syntactical contructs I've ever written.

It is best used in combination with `FindElement` (see prior section on how to shortcut `FindElement`).

Normally to change an element, you "find" it, then call its `update` method. The code usually looks like this, as you saw in the previous section:

```python
window[key].update(new_value)

The code can be further compressed by removing the `.update` characters, resulting in this very compact looking call:

```
window[key](new_value)
```

Yes, that's a valid statement in Python.

What's happening is that the element itself is being called. You can also writing it like this:

```
elem = sg.Text('Some text', key='-TEXT-')
elem('new text value')
```

Side note - you can also call your `window` variable directly. If you "call" it it will actually call `Window.read`.

```
window = sg.Window(....)
event, values = window()


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

It is confusing looking however so when used, it might be a good idea to write a comment at the end of the statement to help out the poor beginner programmer coming along behind you.

Because it's such a foreign construct that someone with 1 week of Python classes will not reconize, the demos will continue to use the `.update` method.

It does not have to be used in conjuction with `FindElement`. The call works on any previously made Element. Sometimes elements are created, stored into a variable and then that variable is used in the layout. For example.

```
graph_element = sg.Graph(...... lots of parms ......)

layout = [[graph_element]]
.
.
.
graph_element(background_color='blue')
```

Hopefully this isn't too confusing. Note that the methods these shortcuts replace will not be removed. You can continue to use the old constructs without changes.

Fonts

Already discussed in the common parameters section. Either string or a tuple.

Color in PySimpleGUI are in one of two formats - color name or RGB value.

Individual colors are specified using either the color names as defined in tkinter or an RGB

string of this format:

"#RRGGBB" or "darkblue"

`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. You need to remember this when you create `Text` elements that you are using for output.

`Text('', key='_TXTOUT_)` will create a `Text` Element that has 0 length. If you try to output a string that's 5 characters, it won't be shown in the window because there isn't enough room. The remedy is to manually set the size to what you expect to output

`Text('', size=(15,1), key='_TXTOUT_)` creates a `Text` Element that can hold 15 characters.

Chortcut functions

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

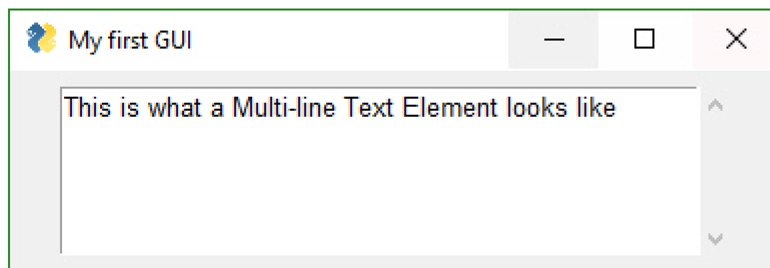
Events `enable_events`

If you set the parameter `enable_events` then you will get an event if the user clicks on the Text.

Multiline Element

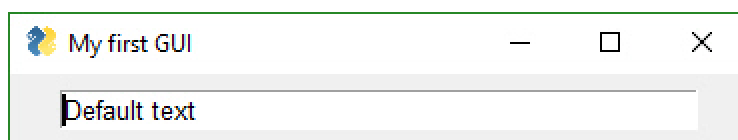
This Element doubles as both an input and output Element.

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



Text Input Element | `InputText == Input == In`

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



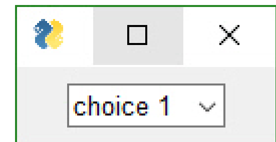
Note about the `do_not_clear` parameter

This used to really trip people up, but don't think so anymore. The `do_not_clear` parameter is initialized when creating the InputText Element. If set to False, then the input field's contents will be erased after every `Window.Read()` call. Use this setting for when your window is an "Input Form" type of window where you want all of the fields to be erased and start over again every time.

Combo Element | `Combo == InputCombo == DropDown == Drop`

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.

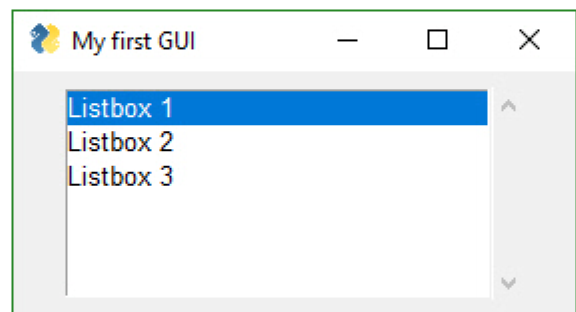
```
layout = [[sg.Combo(['choice 1', 'choice 2'])]]
```



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))]]
```



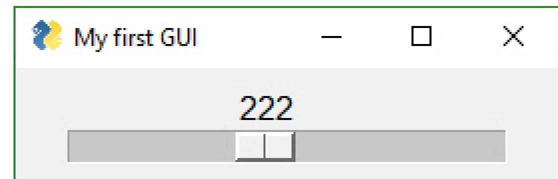
ListBoxes can cause a window to return from a Read call. If the flag `enable_events` is set, then when a user makes a selection, the Read immediately returns.

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))]]
```



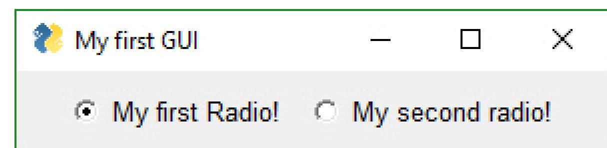
Qt Sliders

There is an important difference between Qt and tkinter sliders. On Qt, the slider values must be integer, not float. If you want your slider to go from 0.1 to 1.0, then make your slider go from 1 to 10 and divide by 10. It's an easy math thing to do and not a big deal. Just deal with it.... you're writing software after all. Presumably you know how to do these things. ;-)

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.

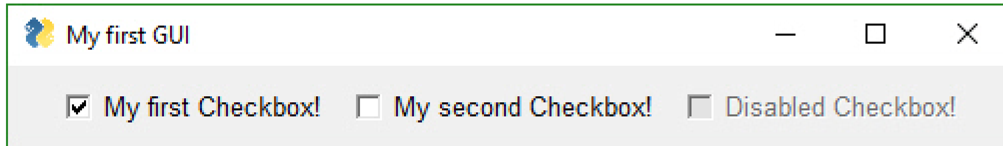
```
layout =  [
    [sg.Radio('My first Radio!', "RADIO1", default=True),
    sg.Radio('My second radio!', "RADIO1")]
]
```



Checkbox Element | `CBox == CB == Check`

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!')]]
```



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')]]
```

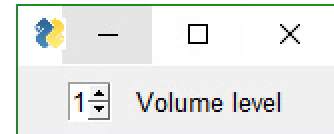
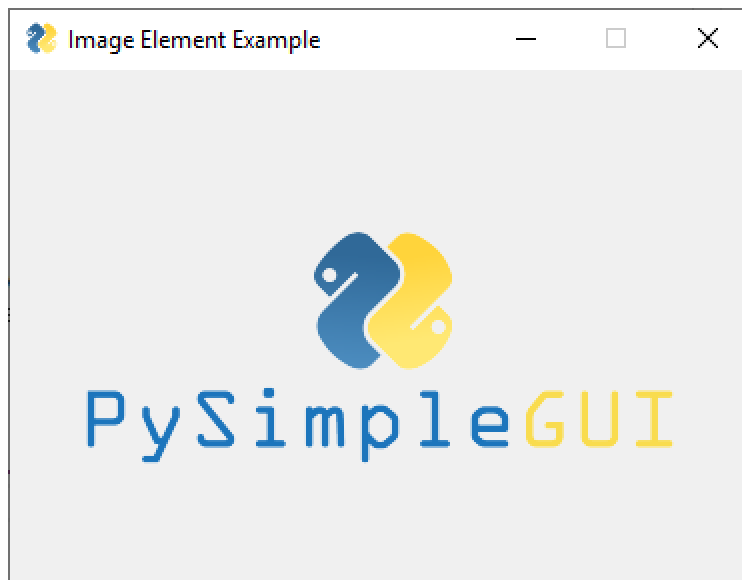


Image Element

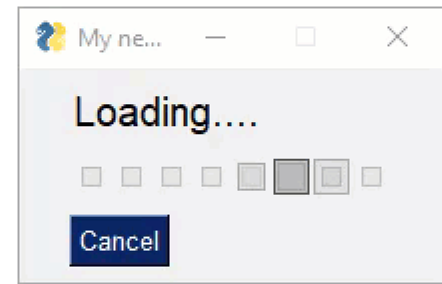
Images can be placed in your window provide they are in PNG, GIF, PPM/PGM format. JPGs cannot be shown because tkinter does not naively support JPGs. You can use the Python Imaging Library (PIL) package to convert your image to PNG prior to calling PySimpleGUI if your images are in JPG format.

```
layout = [
          [sg.Image(r'C:\PySimpleGUI\Logos\PySimpleGUI_Logo_320.png')],
        ]
```



You can specify an animated GIF as an image and can animate the GIF by calling `UpdateAnimation` . Exciting stuff!

You can call the method without setting the `time_between_frames` value and it will show a frame and immediately move on to the next frame. This enables you to do the inter-frame timing.



Button Element

MAC USERS - Macs suck when it comes to tkinter and button colors. It sucks so badly with colors that the `LookAndFeel` call is disabled. You cannot change button colors for Macs. You're stuck with the system default color if you are using the tkinter version of PySimpleGUI. The Qt version does not have this issue.

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, do NOT close the window... they used to. Now to close a window you need to use a CloseButton / CButton.

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), normal Buttons which leave the windows open and CloseButtons that close the window when clicked.

Sometimes there are multiple names for the same function. This is simply to make the job of the programmer quicker and easier. Or they are old names that are no longer used but kept around so that existing programs don't break.

The 4 primary windows of PySimpleGUI buttons and their names are:

1. `Button` = `ReadButton` = `RButton` = `ReadFormButton` (Use `Button`, others are old methods)
2. `CloseButton` = `CButton`
3. `RealtimeButton`
4. `DummyButton`

You will find the long-form names in the older programs. ReadButton for example.

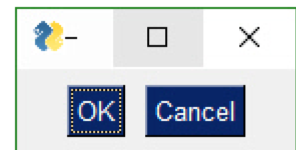
In Oct 2018, the definition of Button changed. Previously Button would CLOSE the window when clicked. It has been changed so the Button calls will leave the window open in exactly the same way as a ReadButton. They are the same calls now. To enables windows to be closed using buttons, a new button was added... `CloseButton` or `CButton`.

Your PySimpleGUI program is most likely going to contain only `Button` calls. The others are generally not foundin user code.

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

```
layout = [[sg.Button('Ok'), sg.Button('Cancel')]]
```

You will rarely see these 2 buttons in particular written this way. Recall that PySimpleGUI is focused on YOU (which generally directly means.... less typing). As a result, the code for the above window is normally written using shortcuts found in the next section.



You will typically see this instead of calls to `Button`:

```
layout = [[sg.Ok(), sg.Cancel()]]
```

In reality `Button` is in fact being called on your behalf. Behind the scenes, `sg.Ok` and `sg.Cancel` call `Button` with the text set to `Ok` and `Cancel` and returning the results that then go into the layout. If you were to print the layout it will look identical to the first layout shown that has `Button` shown specifically in the layout.

Button Element Shortcuts

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
- Open

"Chooser" Buttons

These buttons are used to show dialog boxes that choose something like a filename, date, color, etc. that are filled into an `InputText` Element (or some other "target".... see below regarding targets)

- CalendarButton
- ColorChooserButton
- FileBrowse
- FilesBrowse
- FileSaveAs
- FolderBrowse

IMPORT NOTE ABOUT SHORTCUT BUTTONS Prior to release 3.11.0, these buttons closed the window. Starting with 3.11 they will not close the window. They act like RButtons (return the button text and do not close the window)

If you are having trouble with these buttons closing your window, please check your installed version of PySimpleGUI by typing `pip list` at a command prompt. Prior to 3.11 these buttons close your window.

Using older versions, if you want a Submit() button that does not close the window, then you would instead use RButton('Submit'). Using the new version, if you want a Submit button that closes the window like the sold Submit() call did, you would write that as `CloseButton('Submit')` or `CButton('Submit')`

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 or the button itself. 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.

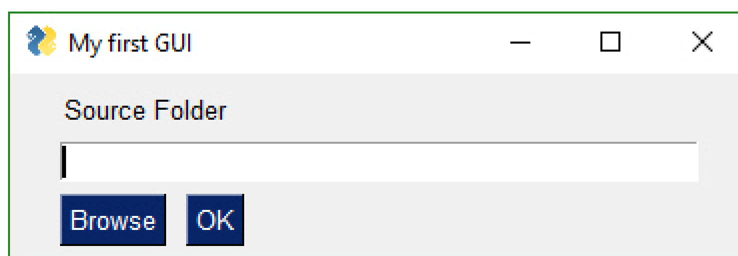
If the Target is specified using (row, column) then it utilizes a grid system. The rows in your GUI are numbered starting with 0. The target can be specified as a hard coded grid item or it can be relative to the button.

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()]]
```

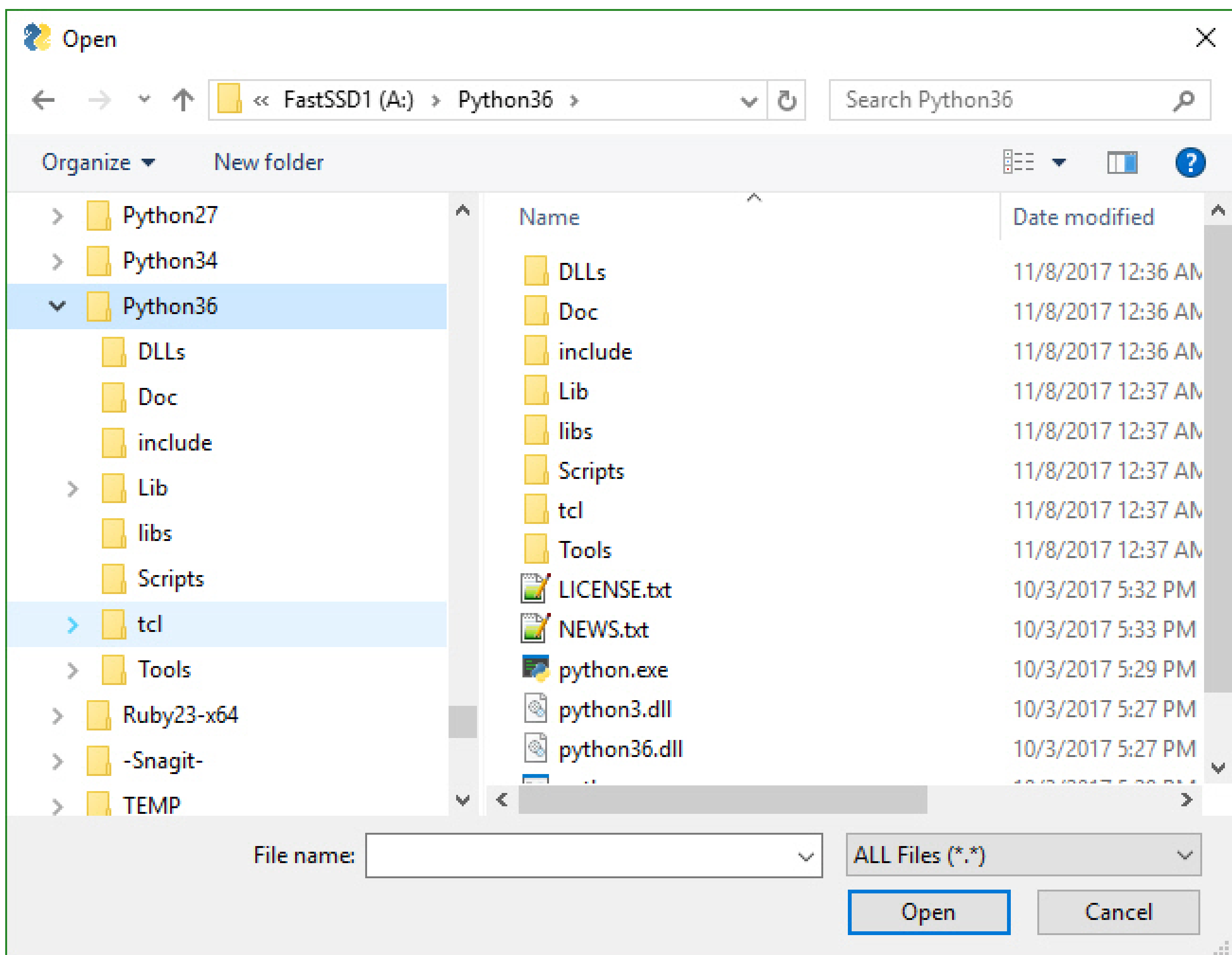
See how much easier the key method is?

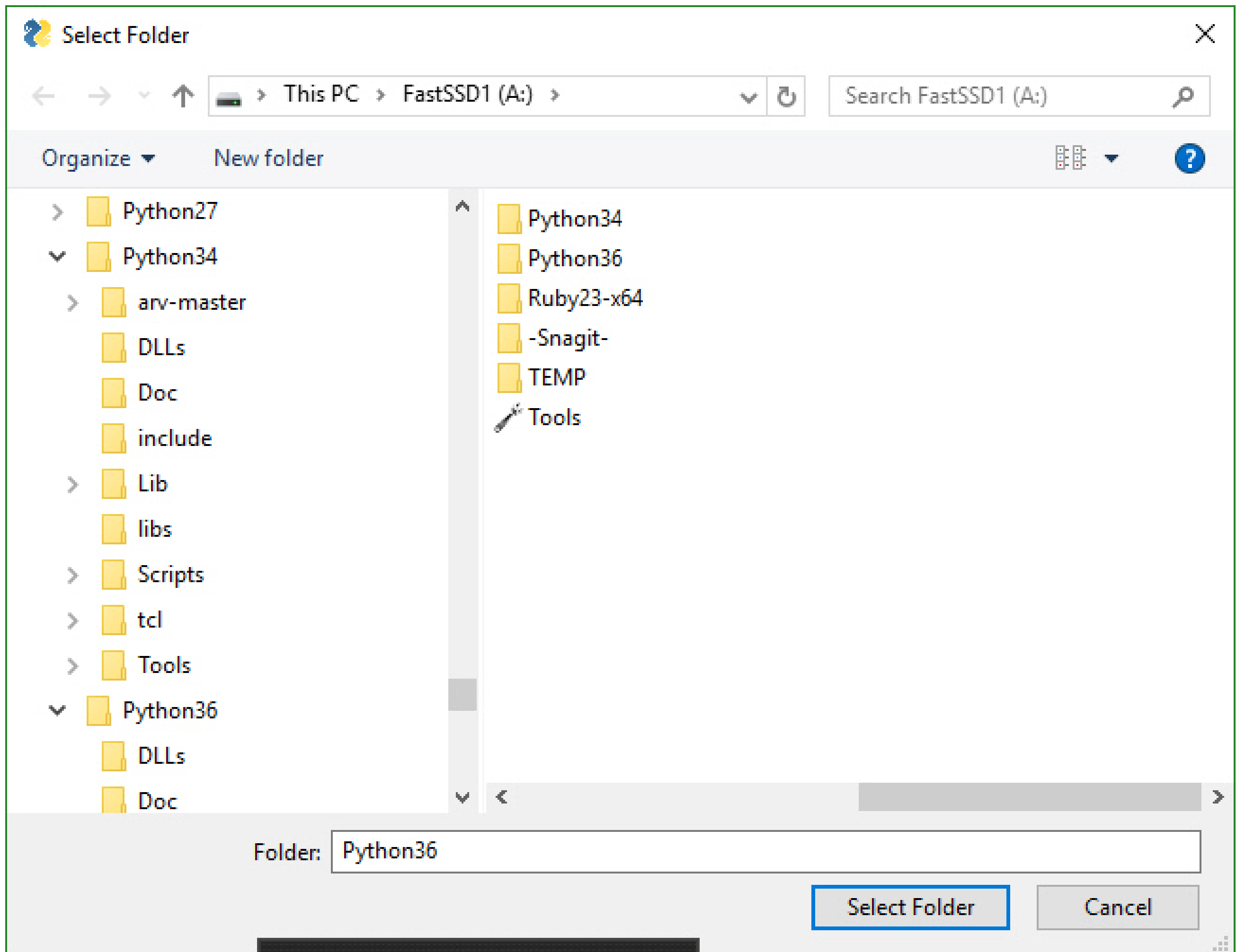
Invisible Targets

One very handy trick is to make your target invisible. This will remove the ability to edit the chosen value like you normally would be able to with an Input Element. It's a way of making things look cleaner, less cluttered too perhaps.

Save & Open Buttons

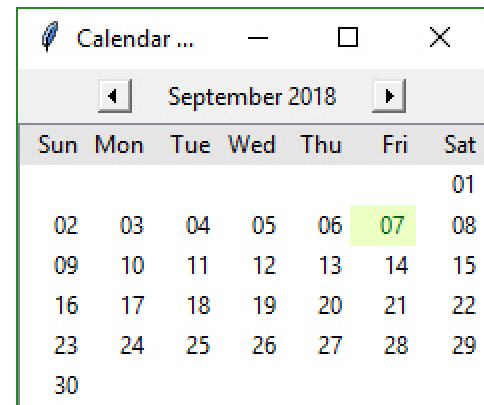
There are 4 different types of File/Folder open dialog box available. If you are looking for a file to open, the `FileBrowse` is what you want. If you want to save a file, `SaveAs` is the button. If you want to get a folder name, then `FolderBrowse` is the button to use. To open several files at once, use the `FilesBrowse` button. It will create a list of files that are separated by ';'




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

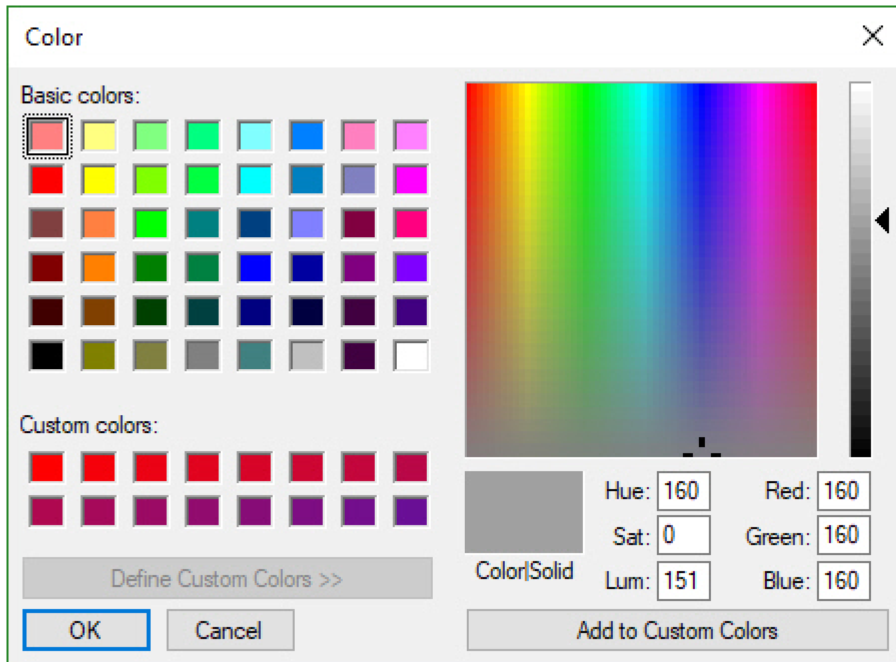
Calendar Buttons

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



Color Chooser Buttons

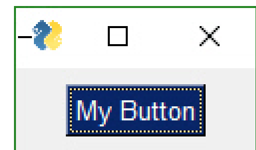
These buttons pop up a standard color chooser window. The result is returned as a tuple. One of the returned values is an RGB hex representation.



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` parameter 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 was clicked. However, you can also use keys on your buttons so that they will be unique. If only the text were used, you would never be able to have 2 buttons in the same window with the same text.

```
layout = [[sg.Button('My Button', key='_BUTTON_KEY_')]]
```

With this layout, the event that is returned from a `Window.Read()` call when the button is clicked will be " `_BUTTON_KEY_` "

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. If you want to set the button background color to the current system default, use the value COLOR_SYSTEM_DEFAULT as the background color.

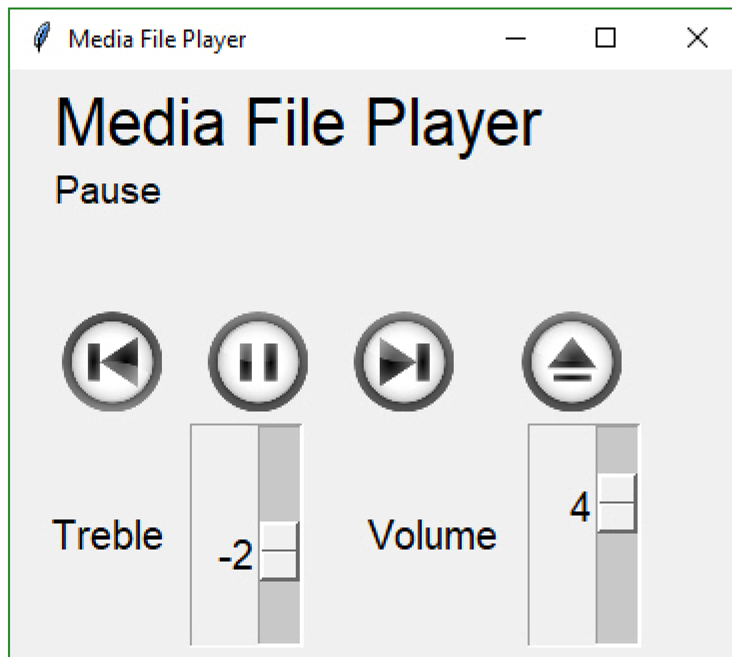
This example comes from the `Demo Media Player.py` example program. Because it's a non-blocking button, it's defined as `RButton` . You also put images on blocking buttons by using `Button` .

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

Experimentation is sometimes required for these concepts to really sink in.

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:

`![robot remote](https://user-images.githubusercontent.com/13696193/44959958-ff9b7000-aec4-11e8-99ea-7450926409be.jpg)`

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

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

```python
import PySimpleGUI as sg

gui_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 = sg.Window('Robotics Remote Control', gui_rows)

#
# Some place later in your code...
# You need to perform a Read or Refresh call on your window every now and then or
# else it will apprear as if the program has locked up.
#
# your program's main loop
while (True):
    # This is the code that reads and updates your window
    event, values = window.Read(timeout=50)
    print(event)
    if event in ('Quit', None):
        break

window.Close()  # Don't forget to close your window!
```

This loop will read button values and print them. When one of the Realtime buttons is clicked, the call to `window.Read` will return a button name matching the name on the button that was depressed or the key if there was a key assigned to the button. It will continue to return values as long as the button remains depressed. Once released, the Read will return timeout events 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"),))]]
```

NOTE - Mac users will not be able to use the file_types parameter. tkinter has a bug on Macs that will crash the program is a file_type is attempted so that feature had to be removed. Sorry about that!

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.

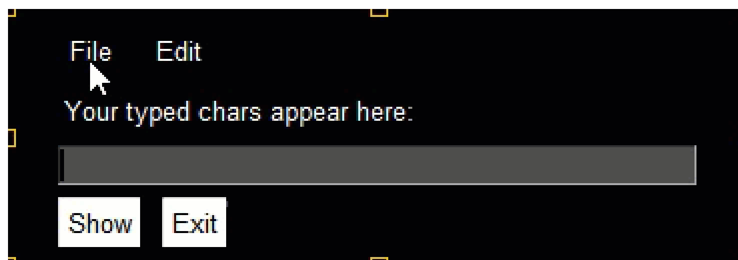
The Enter Key can be "bound" to a particular button so that when the key is pressed, it causes the window to return as if the button was clicked. This is done using the `bind_return_key` parameter in the button calls. If there are more than 1 button on a window, the FIRST button that is of type Close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

The ButtonMenu element produces a unique kind of effect. It's a button, that when clicked, shows you a menu. It's like clicking one of the top-level menu items on a MenuBar. As a result, the menu definition take the format of a single menu entry from a normal menu definition. A normal menu definition is a list of lists. This definition is one of those lists.

```
 ['Menu', ['&Pause Graph', 'Menu item::optional_key']]
```

The very first string normally specifies what is shown on the menu bar. In this case, the value is **not used**. You set the text for the button using a different parameter, the `button_text` parm.

One use of this element is to make a "fake menu bar" that has a colored background. Normal menu bars cannot have their background color changed. Not so with ButtonMenus.



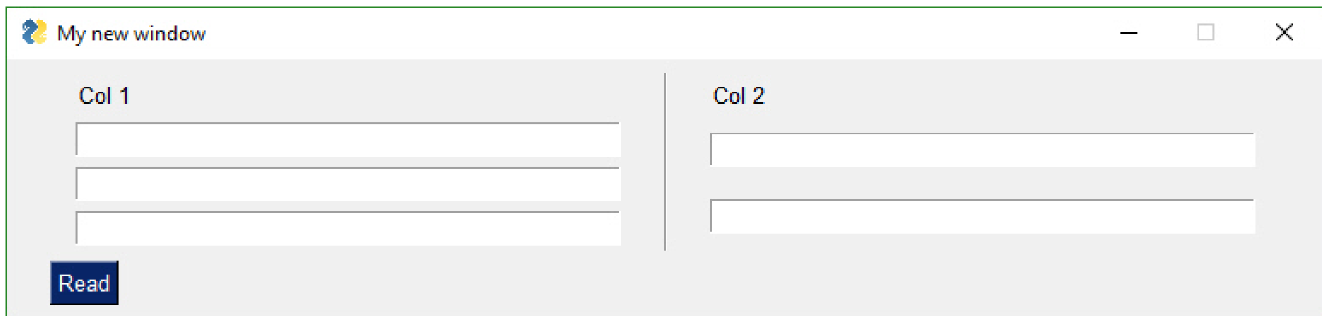
Return values for ButtonMenus are sent via the return values dictionary. If a selection is made, then an event is generated that will equal the ButtonMenu's key value. Use that key value to look up the value selected by the user. This is the same mechanism as the Menu Bar Element, but differs from the pop-up (right click) menu.

VerticalSeparator Element

This element has limited usefulness and is being included more for completeness than anything else. It will draw a line between elements.

It works best when placed between columns or elements that span multiple rows. If on a "normal" row with elements that are only 1 row high, then it will only span that one row.

VerticalSeparator(pad=None)



HorizontalSeparator Element

In PySimpleGUI, the tkinter port, there is no `HorizontalSeparator` Element. One will be added as a "stub" so that code is portable. It will likely do nothing just like the `Stretch` Element.

An easy way to get a horizontal line in PySimpleGUI is to use a `Text` Element that contains a line of underscores

sg.Text('_'*30)

ProgressBar Element

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 Meter 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 = [[sg.Text('A custom progress meter')],
          [sg.ProgressBar(1000, orientation='h', size=(20, 20), key='progressbar')],
          [sg.Cancel()]]

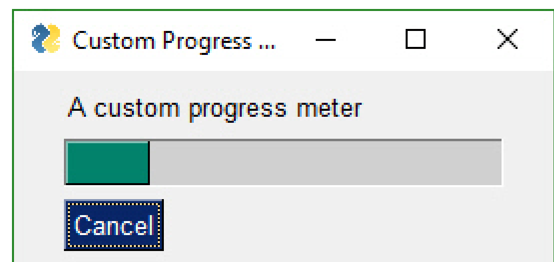

window = sg.Window('Custom Progress Meter', layout)
progress_bar = window['progressbar']

for i in range(1000):

    event, values = window.read(timeout=10)
    if event == 'Cancel'  or event is None:
        break

    progress_bar.UpdateBar(i + 1)

window.close()
```



Output Element

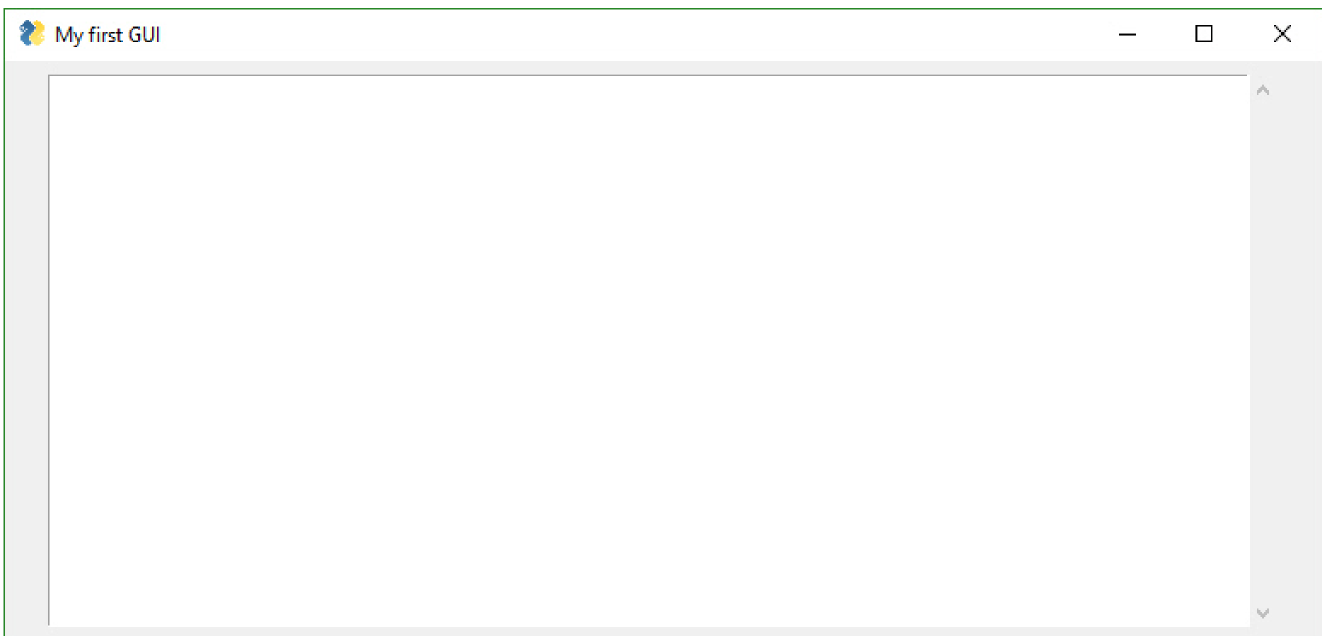
The Output Element is a re-direction of Stdout.

If you are looking for a way to quickly add the ability to show scrolling text within your window, then adding an `Output` Element is about as quick and easy as it gets.

Anything "printed" will be displayed in this element. This is the "trivial" way to show scrolling text in your window. It's as easy as dropping an Output Element into your window and then calling print as much as you want. The user will see a scrolling area of text inside their window.

IMPORTANT You will NOT see what you `print` until you call either `window.Read` or `window.Refresh`. If you want to immediately see what was printed, call `window.Refresh()` immediately after your print statement.

Output(size=(80,20))



Here's a complete solution for a chat-window using an Output Element. To display data that's received, you would to simply "print" it and it will show up in the output area. You'll find this technique used in several Demo Programs including the HowDoI application.

```
import PySimpleGUI as sg

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.Button('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', layout, default_element_size=(30, 2))


    while True:
        event, value = window.read()
        if event == 'SEND':
            print(value)
        else:
            break
    window.close()
ChatBot()
```

Column Element & Frame, Tab "Container" Elements

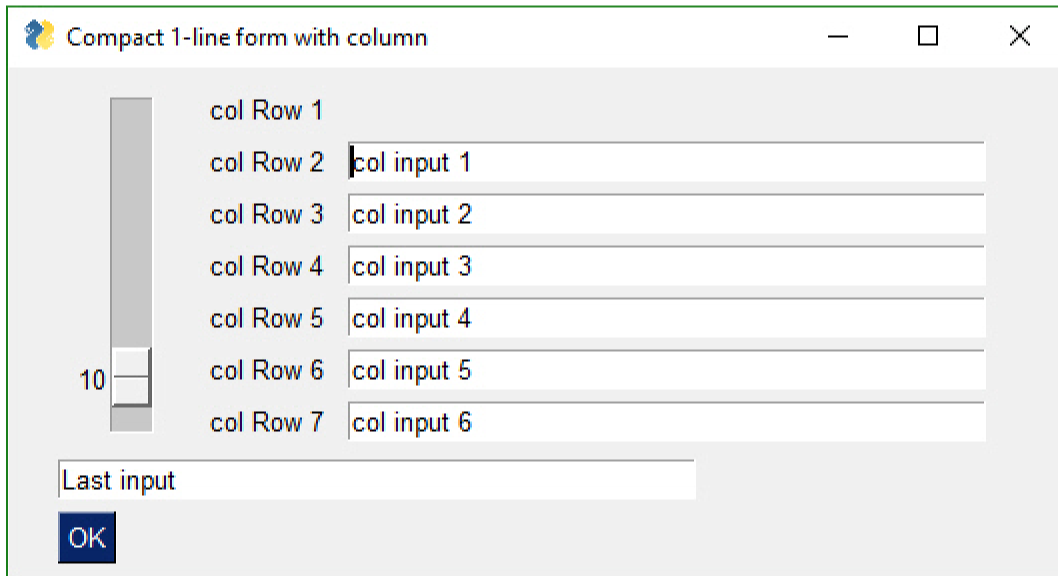
Columns and Frames and Tabs are all "Container Elements" and behave similarly. This section focuses on Columns but can be applied elsewhere.

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, like all "container elements", in exactly the same way as a window, as a list of lists.

Columns are needed when you want to specify more than 1 element in a single row.

For example, this layout has a single slider element that spans several rows followed by 7 `Text` and `Input` elements on the same row.



Without a Column Element you can't create a layout like this. But with it, you should be able to closely match any layout created using tkinter only.

```
import PySimpleGUI as sg




window = sg.Window('Columns')


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()]]



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

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

Column, Frame, Tab, Window element_justification

Beginning in Release 4.3 you can set the justification for any container element. This is done through the `element_justification` parameter. This will greatly help anyone that wants to center all of their content in a window. Previously it was difficult to do these kinds of layouts, if not impossible.

justify the `Column` element's row by setting the `Column` 's `justification` parameter.

You can also justify the entire contents within a `Column` by using the Column's `element_justification` parameter.

With these parameter's it is possible to create windows that have their contents centered. Previously this was very difficult to do.

This is currently only available in the primary PySimpleGUI port.

They can also be used to justify a group of elements in a particular way.

Placing `Column` elements inside `Columns` elements make it possible to create a multitude of

Sizer Element

New in 4.3 is the `Sizer` Element. This element is used to help create a container of a particular size. It can be placed inside of these PySimpleGUI items:

- Column
- Frame
- Tab
- Window

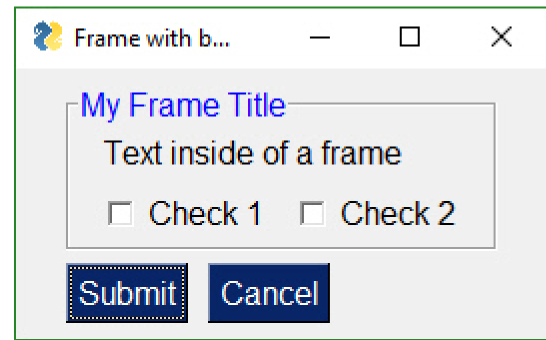
The implementation of a `Sizer` is quite simple. It returns an empty `Column` element that has a pad value set to the values passed into the `Sizer` . Thus isn't not a class but rather a "Shortcut function" similar to the pre-defined Buttons.

This feature is only available in the tkinter port of PySimpleGUI at the moment. A cross port is needed.

Frame Element (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. Like a Column element, it's a "Container Element" that holds one or more elements inside.

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.

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', layout, font=("Helvetica", 12))
```

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.

However, there's another way to get that power and that's through the Graph Element, an even MORE powerful Element as it uses a Canvas that you can directly access if needed. The Graph Element has a large number of drawing methods that the Canvas Element does not have. Plus, if you need to, you can access the Graph Element's "Canvas" through a member variable.

Matplotlib, Pyplot Integration

NOTE - The newest version of Matplotlib (3.1.0) no longer works with this technique. You must install 3.0.3 in order to use the Demo Matplotlib programs provided in the Demo Programs section.

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

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))]]


window = sg.Window('Demo Application - Embedding Matplotlib In PySimpleGUI', layout).Finalize()


fig_photo = draw_figure(window.FindElement('canvas').TKCanvas, fig)


event, 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.

Methods & Properties

TKCanvas - not a method but a property. Returns the tkinter Canvas Widget

Graph Element

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

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.... sometimes... or was it in the lower left? In short, it's a **pain in the ass**.

How about the ability to get your own location of (0,0) and then using those coordinates instead of what tkinter provides? This results in a very powerful capability - working in your own units, and then displaying them in an area defined in pixels.

If you've ever been frustrated with where (0,0) is located on some surface you draw on, then fear not, your frustration ends right here. You get to draw using whatever coordinate system you want. Place (0,0) anywhere you want, including not anywhere on your Graph. You could define a Graph that's all negative numbers between -2.1 and -3.5 in the X axis and -3 to -8.2 in the 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
- DrawImage

You can move your figures around on the canvas by supplying the Figure ID the **x,y delta** to move. It does not move to an absolute position, but rather an offset from where the figure is now. (Use Relocate to move to a specific location)

```
graph.MoveFigure(my_circle, 10, 10)
```

You'll also use this ID to delete individual figures you've drawn:

```
graph.DeleteFigure(my_circle)
```

Mouse Events Inside Graph Elements

If you have eneabled events for your Graph Element, then you can receive mouse click events. If you additionally enable `drag_submits` in your creation of the Graph Element, then you will also get events when you "DRAG" inside of a window. A "Drag" is defined as a left button down and then the mouse is moved.

When a drag event happens, the event will be the Graph Element's key. The `value` returned in the values dictionary is a tuple of the (x,y) location of the mouse currently.

This means you'll get a "stream" of events. If the mouse moves, you'll get at LEAST 1 and likely a lot more than 1 event.

Mouse Up Event for Drags

When you've got `drag_submits` enabled, there's a sticky situation that arises.... what happens when you're done dragging and you've let go of the mouse button? How is the "Mouse Up" event relayed back to your code.

The "Mouse Up" will generate an event to you with the value: `Graph_key` + `'+UP'`. Thus, if your Graph Element has a key of `'_GRAPH_'`, then the event you will receive when the mouse button is released is: `'_GRAPH_+UP'`

Yea, it's a little weird, but it works. It's SIMPLE too. I recommend using the `.startswith` and `.endswith` built-ins when dealing with these kinds of string values.

Here is an example of the `events` and the `values dictionary` that was generated by clicking and dragging inside of a Graph Element with the key == 'graph':

```
graph {'graph': (159, 256)}
graph {'graph': (157, 256)}
graph {'graph': (157, 256)}
graph {'graph': (157, 254)}
graph {'graph': (157, 254)}
graph {'graph': (154, 254)}
graph {'graph': (154, 254)}
graph+UP {'graph': (154, 254)}
```

Table Element

Table and Tree Elements are of the most complex in PySimpleGUI. They have a lot of options and a lot of unusual characteristics.

`window.read()` return values from Table Element

The values returned from a `Window.Read` call for the Table Element are a list of row numbers that are currently highlighted.

The Qt `Table.Get()` call

New in **PySimpleGUIQt** is the addition of the `Table` method `Get`. This method returns the table that is currently being shown in the GUI. This method was required in order to obtain any edits the user may have made to the table.

For the tkinter port, it will return the same values that was passed in when the table was created because tkinter Tables cannot be modified by the user (please file an Issue if you know a way).

Known `Table` visualization problem....

There has been an elusive problem where clicking on or near the table's header caused tkinter to go crazy and resize the columns continuously as you moved the mouse.

This problem has existed since the first release of the `Table` element. It was fixed in release 4.3.

Known table colors in Python 3.7.3, 3.7.4, 3.8, ?

The tkinter that's been released in the past several releases of Python has a bug. Table colors of all types are not working, at all. The background of the rows never change. If that's important to you, you'll need to **downgrade** your Python version. 3.6 works really well with PySimpleGUI and tkinter.

Empty Tables

If you wish to start your table as being an empty one, you will need to specify an empty table. This list comprehension will create an empty table with 15 rows and 6 columns.

```
data = [['' for row in range(15)]for col in range(6)]
```

Events from Tables

There are two ways to get events generated from Table Element.
`change_submits` event generated as soon as a row is clicked on `bind_return_key` event generate when a row is double clicked or the return key is press while on a row.

Tree Element

The Tree Element and Table Element are close cousins. Many of the parameters found in the Table Element apply to Tree Elements. In particular the heading information, column widths, etc.

Unlike Tables there is no standard format for trees. Thus the data structure passed to the Tree Element must be constructed. This is done using the TreeData class. The process is as follows:

- Get a TreeData Object
- "Insert" data into the tree
- Pass the filled in TreeData object to Tree Element

TreeData format

```
def TreeData()
def Insert(self, parent, key, text, values, icon=None)
```

To "insert" data into the tree the TreeData method Insert is called.

```
Insert(parent_key, key, display_text, values)
```

To indicate insertion at the head of the tree, use a parent key of "". So, every top-level node in the tree will have a parent node = ""

This code creates a TreeData object and populates with 3 values

```
treedata = sg.TreeData()

treedata.Insert("", '_A_', 'A', [1,2,3])
treedata.Insert("", '_B_', 'B', [4,5,6])
treedata.Insert("_A_", '_A1_', 'A1', ['can','be','anything'])
```

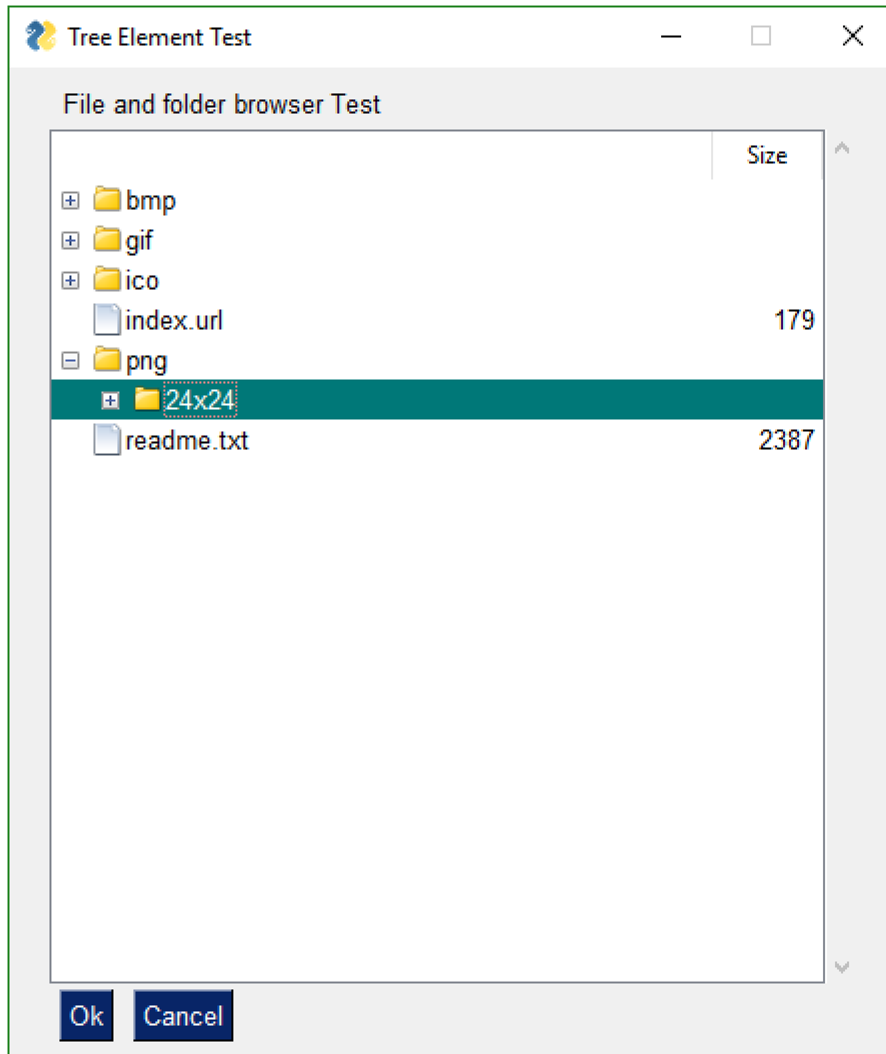
Note that you ***can*** use the same values for display_text and keys. The only thing you have to watch for is that you cannot repeat keys.

When Reading a window the Table Element will return a list of rows that are selected by the user. The list will be empty is no rows are selected.

Icons on Tree Entries

If you wish to show an icon next to a tree item, then you specify the icon in the call to `Insert`. You pass in a filename or a Base64 bytes string using the optional `icon` parameter.

Here is the result of showing an icon with a tree entry.



Tab and Tab Group Elements

Tabs are another of PySimpleGUI "Container Elements". It is capable of "containing" a layout just as a window contains a layout. Other container elements include the `Column` and `Frame` elements.

Just like windows and the other container elements, the `Tab` Element has a layout consisting of any desired combination of Elements in any desired layouts. You can have Tabs inside of Tabs inside of Columns inside of Windows, etc.

`Tab` layouts look exactly like Window layouts, that is they are **a list of lists of Elements**.

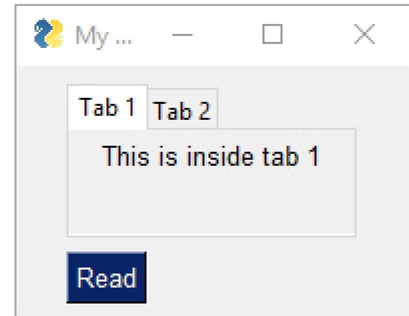
How you place a Tab element into a window is different than all other elements. You cannot place a Tab directly into a Window's layout.

Also, tabs cannot be made invisible at this time. They have a visibily parameter but calling update will not change it.

Tabs are contained in TabGroups. They are **not** placed into other layouts. To get a Tab into your window, first place the `Tab` Element into a `TabGroup` Element and then place the `TabGroup` Element into the Window layout.

Let's look at this Window as an example:

View of second tab:

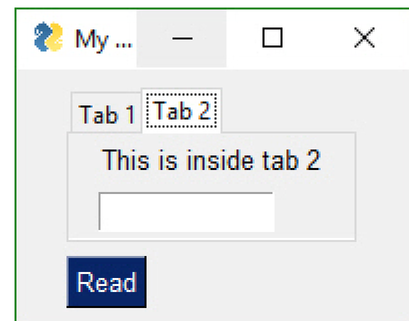



```
tab1_layout =  [[sg.T('This is inside tab 1')]]

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.Button('Read')]]
```



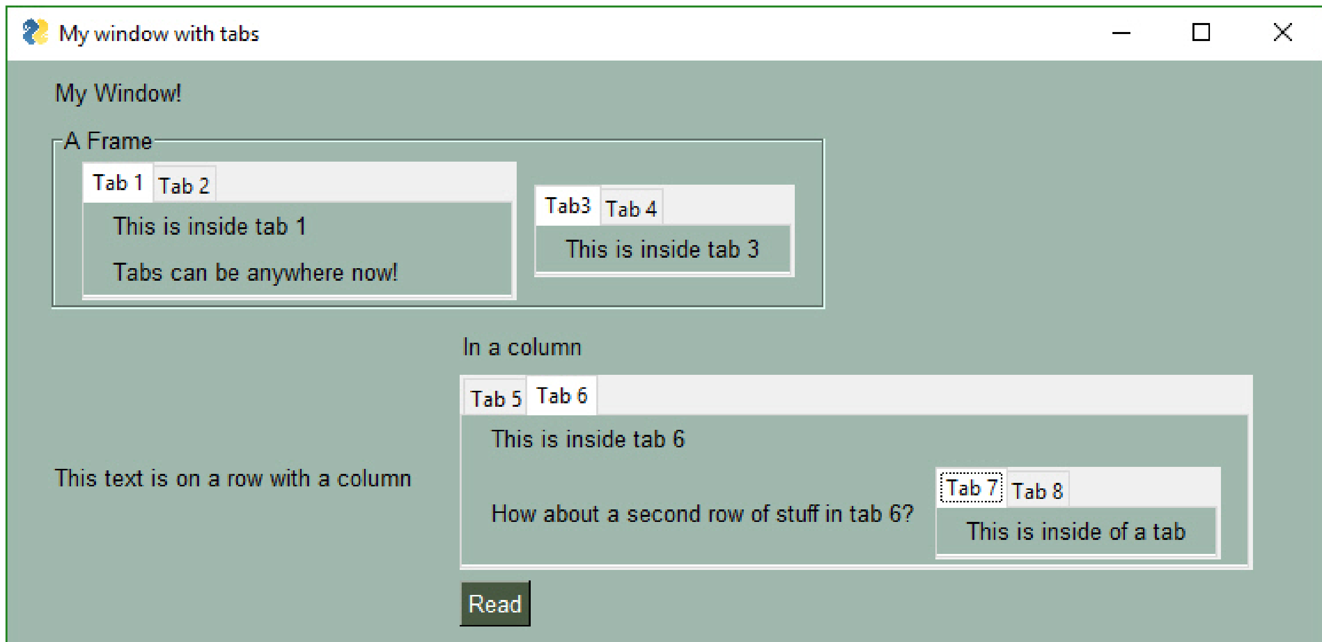
The Window layout has the TabGroup and within the tab Group are the two Tab elements.

One important thing to notice about all of these container Elements and Windows layouts... they all take a "list of lists" as the layout. They all have a layout that looks like this `[[ ]]`

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.

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.

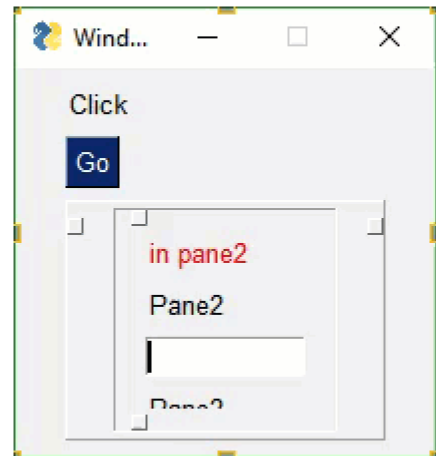
Reading Tab Groups

Tab Groups now return a value when a Read returns. They return which tab is currently selected. There is also a `enable_events` 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

x## Pane Element

New in version 3.20 is the Pane Element, a super-cool tkinter feature. You won't find this one in PySimpleGUIQt, only PySimpleGUI. It's difficult to describe one of these things. Think of them as "Tabs without labels" that you can slide.

Each "Pane" of a Pane Element must be a Column Element. The parameter `pane_list` is a list of Column Elements.



Calls can get a little hairy looking if you try to declare everything in-line as you can see in this example.

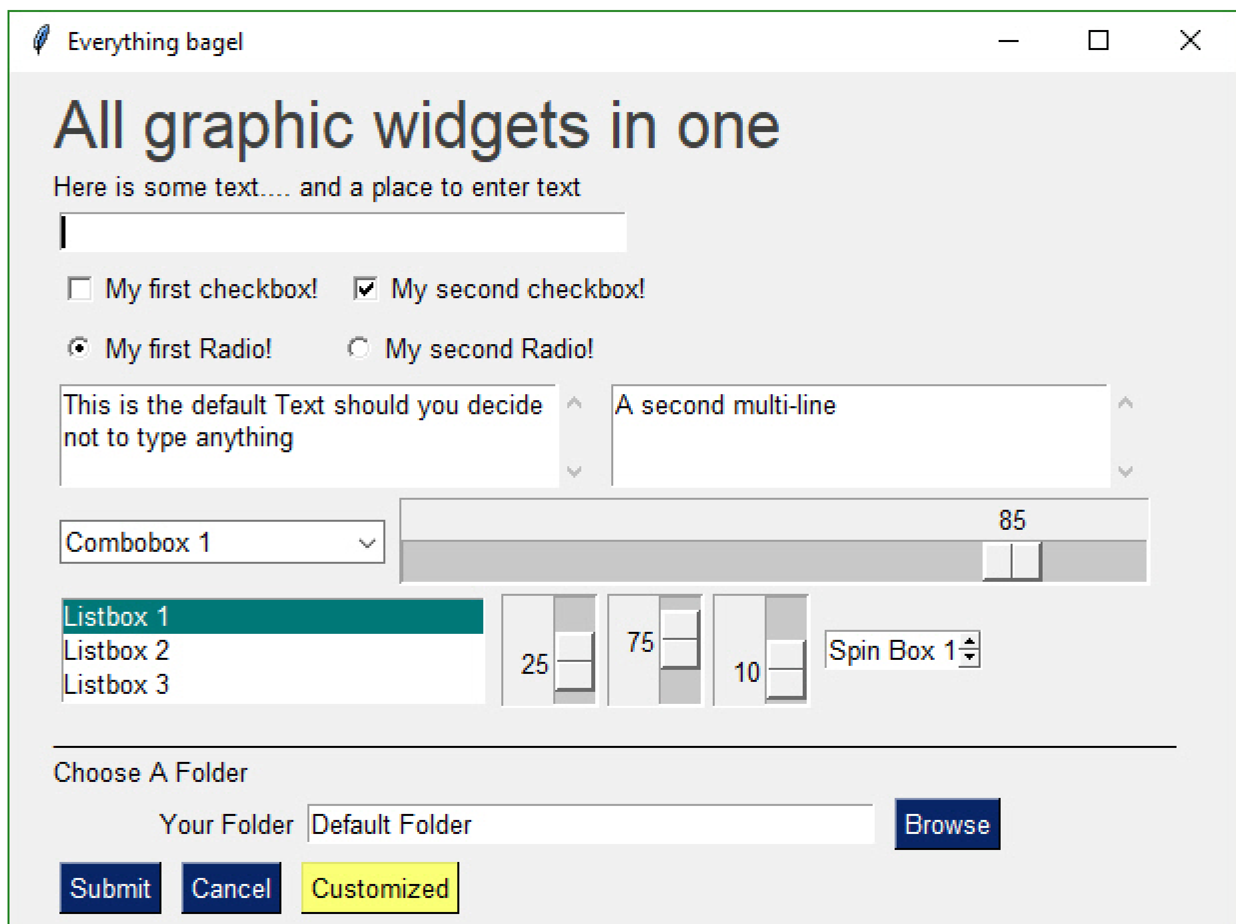
```
sg.Pane([col5, sg.Column([[sg.Pane([col1, col2, col4], handle_size=15,
orientation='v',  background_color=None, show_handle=True,
visible=True, key='_PANE_', border_width=0,
relief=sg.RELIEF_GROOVE),]]),col3 ], orientation='h',
background_color=None, size=(160,160), relief=sg.RELIEF_RAISED,
border_width=0)
```

Combing these with *visibility* make for an interesting interface with entire panes being hidden from view until neded by the user. It's one way of producing "dynamic" windows.

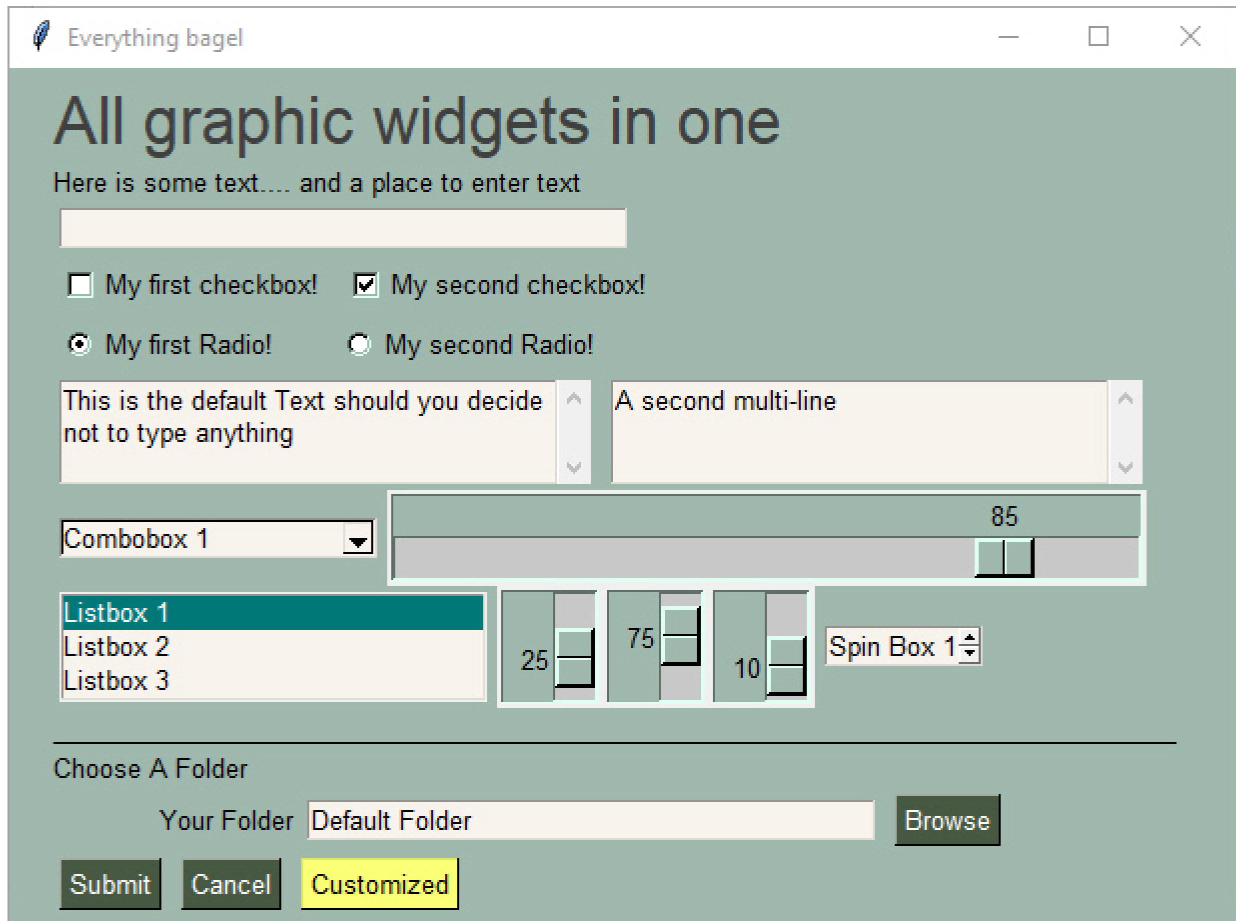
Colors

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'))
```

SystemTray

This is a PySimpleGUIQt and PySimpleGUIWx only feature. Don't know of a way to do it using tkinter. Your source code for SystemTray is identical for the Qt and Wx implementations. You can switch frameworks by simply changing your import statement.

In addition to running normal windows, it's now also possible to have an icon down in the system tray that you can read to get menu events. There is a new SystemTray object that is used much like a Window object. You first get one, then you perform Reads in order to get events.

Here is the definition of the SystemTray object.

```
SystemTray(menu=None, filename=None, data=None, data_base64=None, tooltip=None):
        '''
 SystemTray - create an icon in the system tray
 :param menu: Menu definition
 :param filename: filename for icon
 :param data: in-ram image for icon
 :param data_base64: basee-64 data for icon
 :param tooltip: tooltip string '''
```

You'll notice that there are 3 different ways to specify the icon image. The base-64 parameter allows you to define a variable in your .py code that is the encoded image so that you do not need any additional files. Very handy feature.

System Tray Design Pattern

Here is a design pattern you can use to get a jump-start.

This program will create a system tray icon and perform a blocking Read. If the item "Open" is chosen from the system tray, then a popup is shown.

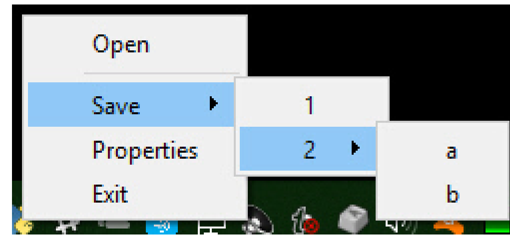
```
import PySimpleGUIQt as sg

menu_def = ['BLANK', ['&Open', '---', '&Save', ['1', '2', ['a', 'b']], '&Properties', 'E&xit']]

tray = sg.SystemTray(menu=menu_def, filename=r'default_icon.ico')

while True:
  menu_item = tray.Read()
    print(menu_item)
    if menu_item == 'Exit':
        break
    elif menu_item == 'Open':
        sg.Popup('Menu item chosen', menu_item)
```

The design pattern creates an icon that will display this menu:



Icons

When specifying "icons", you can use 3 different formats. * `filename` - filename * `data_base64` - base64 byte string * ' `data` - in-ram bitmap or other "raw" image

You will find 3 parameters used to specify these 3 options on both the initialize statement and on the Update method.

```
menu_def = ['BLANK', ['&Open', '&Save', ['1', '2', ['a', 'b']], '!&Properties', 'E&xit']]
```

A menu is defined using a list. A "Menu entry" is a string that specifies: * text shown * keyboard shortcut * key

See section on Menu Keys for more information on using keys with menus.

An entry without a key and keyboard shortcut is a simple string `'Menu Item'`

If you want to make the "M" be a keyboard shortcut, place an `&` in front of the letter that is the shortcut. `'&Menu Item'`

You can add "keys" to make menu items unique or as another way of identifying a menu item than the text shown. The key is added to the text portion by placing `::` after the text.

`'Menu Item::key'`

The first entry can be ignored. `'BLANK` ' was chosen for this example. It's this way because normally you would specify these menus under some heading on a menu-bar. But here there is no heading so it's filled in with any value you want.

Separators If you want a separator between 2 items, add the entry `'---'` and it will add a separator item at that place in your menu.

Disabled menu entries

If you want to disable a menu entry, place a `!` before the menu entry

SystemTray Methods

```
def Read(timeout=None)
    '''
 Reads the context menu
 :param timeout: Optional.  Any value other than None indicates a non-blocking read
 :return:   String representing meny item chosen. None if nothing read.
    '''
```

The `timeout` parameter specifies how long to wait for an event to take place. If nothing happens within the timeout period, then a "timeout event" is returned. These types of reads make it possible to run asynchronously. To run non-blocked, specify `timeout=0` on the Read call.

Read returns the menu text, complete with key, for the menu item chosen. If you specified `Open::key` as the menu entry, and the user clicked on `Open`, then you will receive the string `Open::key` upon completion of the Read.

Read special return values

In addition to Menu Items, the Read call can return several special values. They include:

EVENT_SYSTEM_TRAY_ICON_DOUBLE_CLICKED - Tray icon was double clicked
EVENT_SYSTEM_TRAY_ICON_ACTIVATED - Tray icon was single clicked
EVENT_SYSTEM_TRAY_MESSAGE_CLICKED - a message balloon was clicked TIMEOUT_KEY is returned if no events are available if the timeout value is set in the Read call

Hide

Hides the icon. Note that no message balloons are shown while an icon is hidden.

```
def Hide()
```

Close

Does the same thing as hide

```
def Close()
```

UnHide

Shows a previously hidden icon

```
def UnHide()
```

ShowMessage

Shows a balloon above the icon in the system tray area. You can specify your own icon to be shown in the balloon, or you can set `messageicon` to one of the preset values.

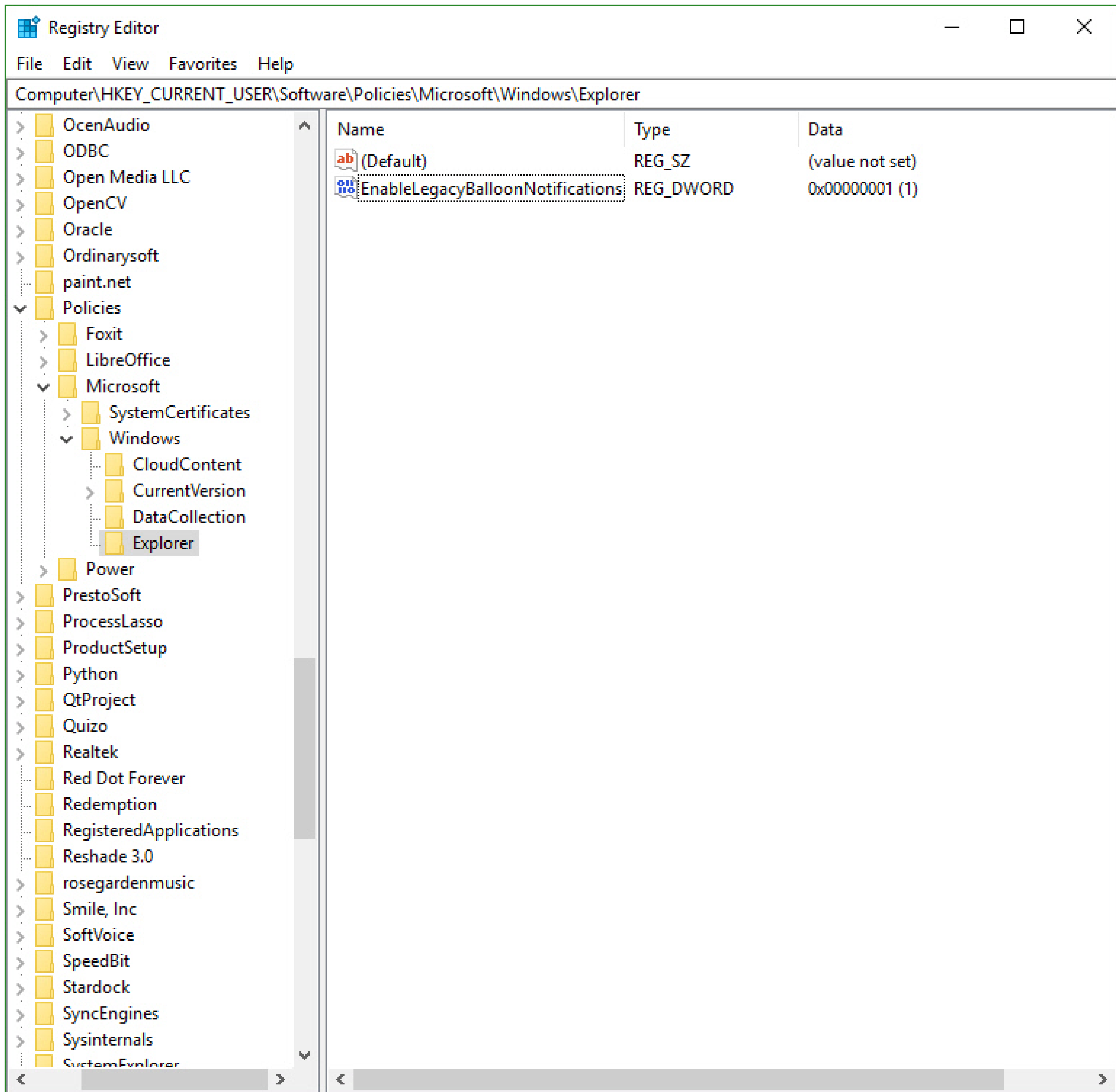
This message has a custom icon.

The preset `messageicon` values are:

```
SYSTEM_TRAY_MESSAGE_ICON_INFORMATION
SYSTEM_TRAY_MESSAGE_ICON_WARNING
SYSTEM_TRAY_MESSAGE_ICON_CRITICAL
SYSTEM_TRAY_MESSAGE_ICON_NOICON
```

```
ShowMessage(title, message, filename=None, data=None,
data_base64=None, messageicon=None, time=10000):
'''
 Shows a balloon above icon in system tray
 :param title:  Title shown in balloon
 :param message: Message to be displayed
 :param filename: Optional icon filename
 :param data: Optional in-ram icon
 :param data_base64: Optional base64 icon
 :param time: How long to display message in milliseconds  :return:
 '''
```

Note, on windows it may be necessary to make a registry change to enable message balloons to be seen. To fix this, you must create the DWORD you see in this screenshot.



Update

You can update any of these items within a SystemTray object * Menu definition * Icon * Tooltip

Change them all or just 1.

Global Settings

There are multiple ways to customize PySimpleGUI. The call with the most granularity (allows access to specific and precise settings). The `ChangeLookAndFeel` call is in reality a single call to `SetOptions` where it changes 13 different settings.

Mac Users - You can't call `ChangeLookAndFeel` but you can call `SetOptions` with any sets of values you want. Nothing is being blocked or filtered.

These settings apply to all windows that are created in the future.

`SetOptions` . The 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:

- Global
- Window
- Element

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)

Apologies that the next few pages are perhaps confusing. There have been a number of changes recently in PySimpleGUI's Read calls that added some really cool stuff, but at the expense of being not so simple. Part of the issue is an attempt to make sure existing code doesn't break. These changes are all in the area of non-blocking reads and reads with timeouts.

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 normal `Read` call with no parameters will block, a call with a `timeout` value of zero will not block.

Note that `InputText` and `MultiLine` Elements will be **cleared** when performing a `Read` . If you do not want your input field to be cleared after a `Read` then you can set the `do_not_clear` parameter to True when creating those elements. The clear is turned on and off on an element by element basis.

The reasoning behind this is that Persistent Windows are often "forms". When "submitting" a form you want to have all of the fields left blank so the next entry of data will start with a fresh window. Also, when implementing a "Chat Window" type of interface, after each read / send of

the chat data, you want the input field cleared. Think of it as a Texting application. Would you want to have to clear your previous text if you want to send a second text?

The design pattern for Persistent Windows was already shown to you earlier in the document... here it is for your convenience.

```
import PySimpleGUI as sg

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

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

while True:
    event, values = window.read()
    if event is None or event == 'Exit':
        break
    print(event, values)

window.Close()
```

Read(timeout = t, timeout_key=TIMEOUT_KEY)

Read with a timeout is a very good thing for your GUIs to use in a read non-blocking situation, you can use them. If your device can wait for a little while, then use this kind of read. The longer you're able to add to the timeout value, the less CPU time you'll be taking.

One way of thinking of reads with timeouts:

> During the timeout time, you are "yielding" the processor to do other tasks.

But it gets better than just being a good citizen....**your GUI will be more responsive than if you used a non-blocking read**

Let's say you had a device that you want to "poll" every 100ms. The "easy way out" and the only way out until recently was this:

```
while True:
    event, values = window.ReadNonBlocking()
    read_my_hardware()
    time.sleep(.1)
```

This program will quickly test for user input, then deal with the hardware. Then it'll sleep for 100ms, while your gui is non-responsive, then it'll check in with your GUI again. I fully realize this is a crude way of doing things. We're talking dirt simple stuff without trying to use threads, etc to 'get it right'. It's for demonstration purposes.

The new and better way.... using the Read Timeout mechanism, the sleep goes away.

```
while True:
    event, values = window.Read(timeout = 100)
    read_my_hardware()
```

This event loop will run every 100 ms. You're making a Read call, so anything that the use does will return back to you immediately, and you're waiting up to 100ms for the user to do something. If the user doesn't do anything, then the read will timeout and execution will return to the program.

Non-Blocking Windows (Asynchronous reads, timeouts)

You can easily spot a non-blocking call in PySimpleGUI. If you see a call to `Window.Read()` with a timeout parameter set to a value other than `None`, then it is a non-blocking call.

This call to read is asynchronous as it has a timeout value:

```
The new way
```python
event, values = sg.Read(timeout=20)
```

You should use the new way if you're reading this for the first time.

The difference in the 2 calls is in the value of event. For ReadNonBlocking, event will be `None` if there are no other events to report. There is a "problem" with this however. With normal Read calls, an event value of None signified the window was closed. For ReadNonBlocking, the way a closed window is returned is via the values variable being set to None.

sg.TIMEOUT_KEY

If you're using the new, timeout=0 method, then an event value of None signifies that the window was closed, just like a normal Read. That leaves the question of what it is set to when not other events are happening. This value will be the value of `timeout_key`. If you did not specify a timeout_key value in your call to read, then it will be set to a default value of: `TIMEOUT_KEY = __timeout__`

If you wanted to test for "no event" in your loop, it would be written like this:

```
while True:
    event, value = window.Read(timeout=0)
    if event is None:
        break
    if event == sg.TIMEOUT_KEY:
        print("Nothing happened")
```
```

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 timeout in your Read and yuou've got nothing else your program will block on, then you will eat up 100% of the CPU time. It's important to be a good citizen. Don't chew up CPU cycles needlessly. Sometimes your mouse wants to move ya know?

Non-blocking (timeout=0) is generally reserved as a "last resort". Too many times people use non-blocking reads when a blocking read will do just fine.

Small Timeout Values (under 10ms)

Do Not use a timeout of less than 10ms. Otherwise you will simply thrash, spending your time trying to do some GUI stuff, only to be interruped by a timeout timer before it can get anything done. The results are potentially disasterous.

There is a hybrid approach... a read with a timeout. You'll score much higher points on the impressive meter if you're able to use a lot less CPU time by using this type of read.

The most legit time to use a non-blocking window is when you're working directly with hardware. Maybe you're driving a serial bus. If you look at the Event Loop in the Demo_OpenCV_Webcam.py program, you'll see that the read is a non-blocking read. However, there is a place in the event loop where blocking occurs. The point in the loop where you will block is the call to read frames from the webcam. When a frame is available you want to quickly deliver it to the output device, so you don't want your GUI blocking. You want the read from the hardware to block.

Another example can be found in the demo for controlling a robot on a Raspberry Pi. In that application you want to read the direction buttons, forward, backward, etc, and immediately take action. If you are using RealtimeButtons, your only option at the moment is to use non-blocking windows. You have to set the timeout to zero if you want the buttons to be real-time responsive.

However, with these buttons, adding a sleep to your event loop will at least give other processes time to execute. It will, however, starve your GUI. The entire time you're sleeping, your GUI isn't executing.

Periodically Calling `Read`

Let's say you do end up using non-blocking reads... then you've got some housekeeping to do. It's up to you to periodically "refresh" 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.Read` on a periodic basis. Several times a second or more will produce a reasonably snappy GUI.

Exiting (Closing) a Persistent Window

If your window has a button that closes the window, then PySimpleGUI will automatically close the window for you. If all of your buttons are ReadButtons, then it'll be up to you to close the window when done. To close a window, call the `Close` method.

```
window.Close()
```

Persistent Window 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


sg.ChangeLookAndFeel('Black')
sg.SetOptions(element_padding=(0, 0))

layout = [[sg.Text('')],
        [sg.Text('', size=(8, 2), font=('Helvetica', 20), justification='center', key='text')],
        [sg.ReadButton('Pause', key='button', button_color=('white', '#001480')),
         sg.ReadButton('Reset', button_color=('white', '#007339'), key='Reset'),
         sg.Exit(button_color=('white', 'firebrick4'), key='Exit')]]

window = sg.Window('Running Timer', layout, no_titlebar=True, auto_size_buttons=False,
keep_on_top=True, grab_anywhere=True)


current_time = 0
paused = False
start_time = int(round(time.time() * 100))
while (True):

    event, values = window.Read(timeout=10)
    current_time = int(round(time.time() * 100)) - start_time

    window.FindElement('text').Update('{:02d}:{:02d}.{:02d}'.format((current_time // 100) // 60,
                                                              (current_time // 100) % 60,
                                                              current_time % 100))
```

Previously this program was implemented using a sleep in the loop to control the clock tick. This version uses the new timeout parameter. The result is a window that reacts quicker then the one with the sleep and the accuracy is just as good.

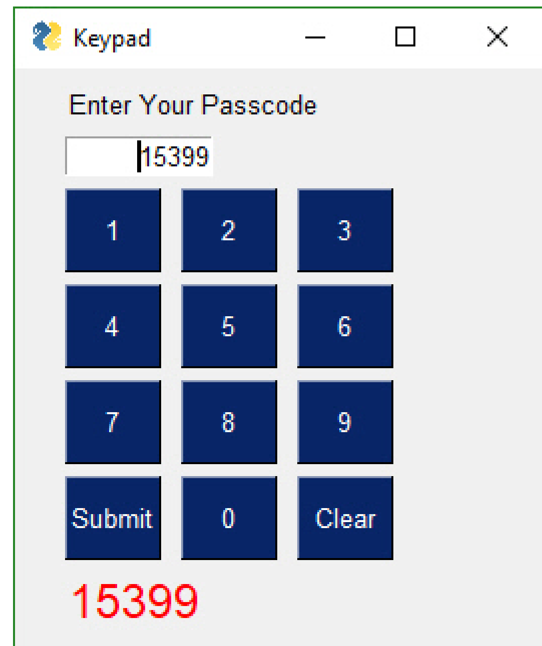
Instead of a Non-blocking Read --- Use `enable_events = True` or `return_keyboard_events = True`

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



Updating Elements (changing element's values in an active window)

If you want to change an Element's settings in your window after the window has been created, then you will call the Element's Update method.

NOTE a window **must be Read or Finalized** before any Update calls can be made. Also, not all settings available to you when you created the Element are available to you via its `Update` method.

Here is an example of updating a Text Element

```
import PySimpleGUI as sg

layout = [ [sg.Text('My layout', key='_TEXT_')],
           [sg.Button('Read')]]

window = sg.Window('My new window', layout)

while True:
    event, values = window.read()
    if event is None:
        break
    window.Element('_TEXT_').Update('My new text value')
```

Notice the placement of the Update call. If you wanted to Update the Text Element *prior* to the Read call, outside of the event loop, then you must call Finalize on the window first.

In this example, the Update is done prior the Read. Because of this, the Finalize call is added to the Window creation.

```
import PySimpleGUI as sg

layout = [ [sg.Text('My layout', key='_TEXT_')],
         [sg.Button('Read')]
       ]

window = sg.Window('My new window', layout).Finalize()

window.Element('_TEXT_').Update('My new text value')

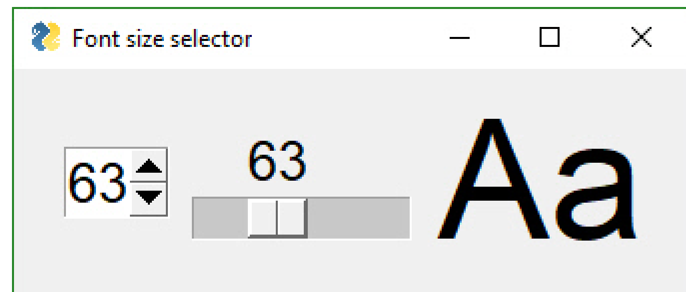
while True:
  event, values = window.read()
    if event is None:
        break
```

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).

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

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.



```
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", layout, grab_anywhere=False)

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

Locating Elements (FindElement == Element == Elem)

The Window method call that's used to find an element is: `FindElement` or the shortened version `Element` or even shorter (version 4.1+) `Elem`

When you see a call to window.FindElement or window.Element, then you know an element is being addressed. Normally this is done so you can call the element's Update method.

ProgressBar / Progress Meters

Note that to change a progress meter's progress, you call `UpdateBar`, not `Update`.

Keyboard & Mouse Capture

NOTE - keyboard capture is currently formatted uniquely among the ports. For basic letters and numbers there is no great differences, but when you start adding Shift and Control or special keyus, they all behave slightly differently. Your best bet is to simply print what is being returned to you to determine what the format for the particular port is.

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



text_elem = sg.Text("", size=(18, 1))

layout = [[sg.Text("Press a key or scroll mouse")],
          [text_elem],
          [sg.Button("OK")]]

window = sg.Window("Keyboard Test", layout,  return_keyboard_events=True, use_default_focus=False)

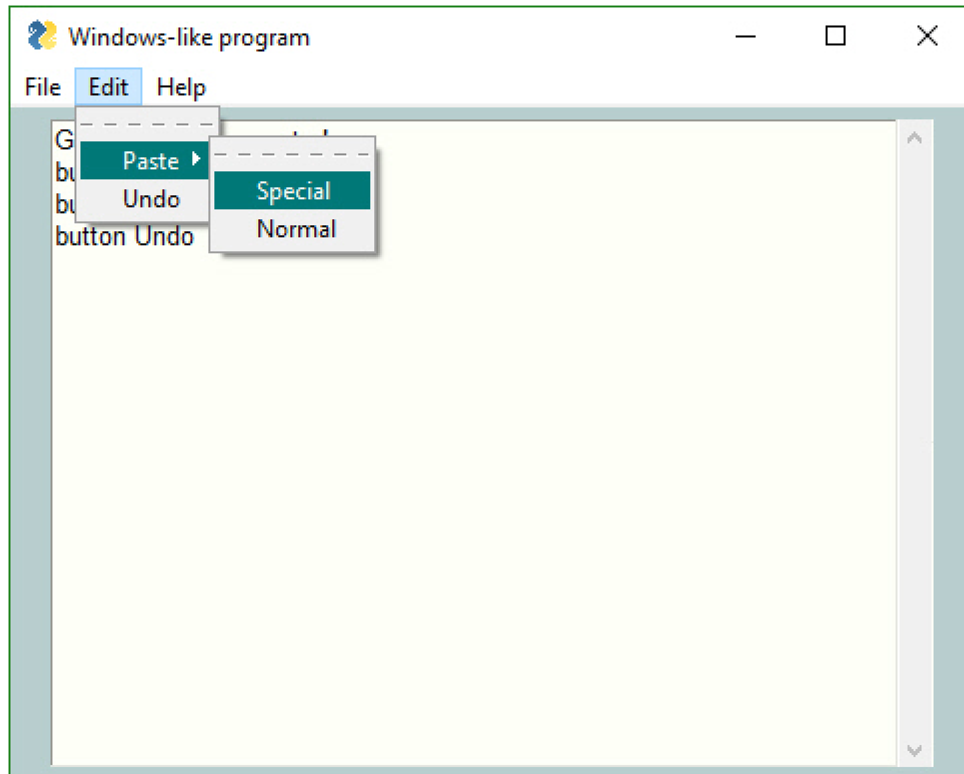

while True:
    event, value = window.read()

    if event == "OK" or event is None:
        print(event, "exiting")
        break
    text_elem.Update(event)
```

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

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 events and as of 3.17, also as in the values dictionary. The value returned will be the entire menu entry, including the key if you specified one.

```
    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.

Methods

To add a menu to a Window place the `Menu` or `MenuBar` element into your layout.

layout = [[sg.Menu(menu_def)]]

It doesn't really matter where you place the Menu Element in your layout as it will always be located at the top of the window.

When the user selects an item, it's returns as the event (along with the menu item's key if one was specified in the menu definition)

Button menus were introduced in version 3.21, having been previously released in PySimpleGUIQt. They work exactly the same and are source code compatible between PySimpleGUI and PySimpleGUIQt. These types of menus take a single menu entry where a Menu Bar takes a list of menu entries.

Return values for ButtonMenus are different than Menu Bars.

You will get back the ButtonMenu's KEY as the event. To get the actual item selected, you will look it up in the values dictionary. This can be done with the expression `values[event]`

Right Click Menus were introduced in version 3.21. Almost every element has a right_click_menu parameter and there is a window-level setting for rich click menu that will attach a right click menu to all elements in the window.

The menu definition is the same as the button menu definition, a single menu entry.

```
right_click_menu = ['&Right', ['Right', '!&Click', '&Menu', 'E&xit', 'Properties']]
```

The first string in a right click menu and a button menu is ***ignored***. It is not used. Normally you would put the string that is shown on the menu bar in that location.

Return values for right click menus are the same as MenuBars. The value chosen is returned as the event.

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.

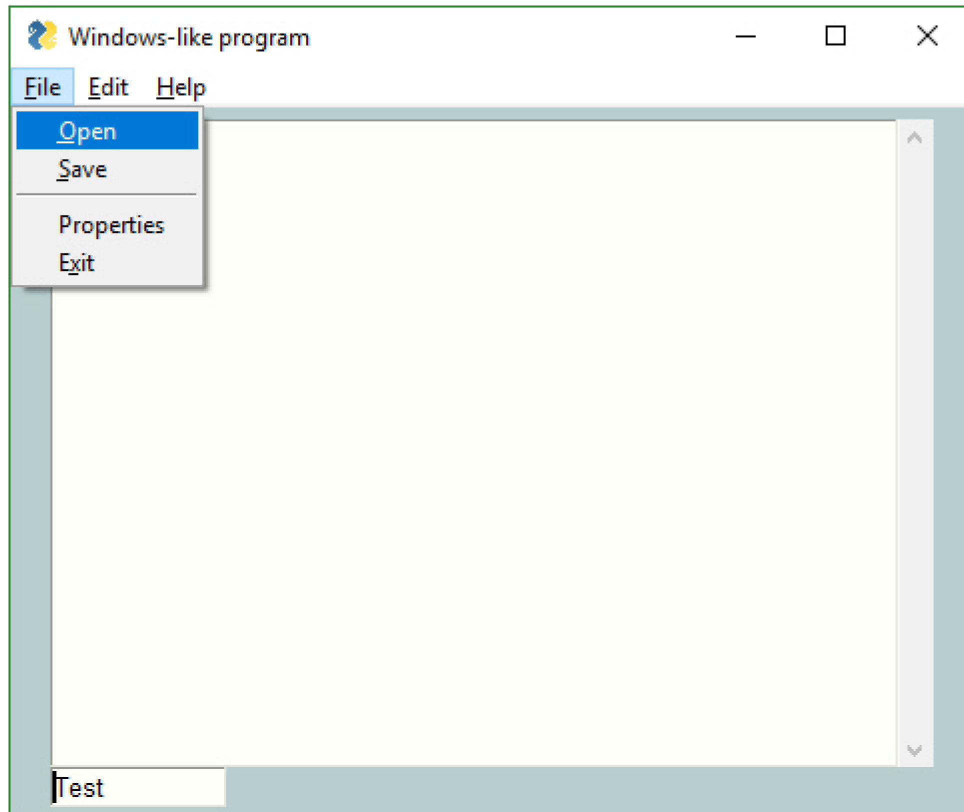
All that's required is for your to add an "&" in front of the letter you want to appear with an underscore. When you hold the Alt key down you will see the menu with underlines that you marked.

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:



If you want one of your menu items to be disabled, then place a '!' in front of the menu entry. To disable the Paste menu entry in the previous examples, the entry would be: `['!&Edit', ['Paste', ['Special', 'Normal',], 'Undo'],]`

If your want to change the disabled menu item flag / character from '!' to something else, change the variable `MENU_DISABLED_CHARACTER`

Beginning in version 3.17 you can add a `key` to your menu entries. The `key` value will be removed prior to be inserted into the menu. When you receive Menu events, the entire menu entry, including the `key` is returned. A key is indicated by adding `::` after a menu entry, followed by the key.

To add the `key` `_MY_KEY_` to the Special menu entry, the code would be:

```
['&Edit', ['Paste', ['Special::_MY_KEY_', 'Normal',], 'Undo'],]
```

If you want to change the characters that indicate a key follows from '::' to something else, change the variable `MENU_KEY_SEPARATOR`

Having read through the Menu section, you may have noticed that the right click menu and the button menu have a format that is a little odd as there is a part of it that is not utilized (the first very string). Perhaps the words "Not Used" should be in the examples.... But, there's a reason to retain words there that make sense.

The reason for this is an architectural one, but it also has a convienence for the user. You can put the individual menu items (button and right click) into a list and you'll have a menu bar definition.

This would work to make a menu bar from a series of these individual menu defintions:

```
menu_bar = [right_click_menu_1, right_click_menu_2, button_menu_def ]
```

And, of course, the direction works the opposite too. You can take a Menu Bar definition and pull out an individual menu item to create a right click or button menu.

Running Multiple Windows

This is where PySimpleGUI continues to be simple, but the problem space just went into the realm of "Complex".

If you wish to run multiple windows in your event loop, then there are 2 methods for doing this.

1. First window does not remain active while second window is visible
2. First window remains active while second window is visible

You will find the 2 design matters in 2 demo programs in the Demo Program area of the GitHub (http://www.PySimpleGUI.com)

Critically important When creating a new window you must use a "fresh" layout every time. You cannot reuse a layout from a previous window. As a result you will see the layout for window 2 being defined inside of the larger event loop.

If you have a window layout that you used with a window and you've closed the window, you cannot use the specific elements that were in that window. You must RE-CREATE your `layout` variable every time you create a new window. Read that phrase again.... You must RE-CREATE your `layout` variable every time you create a new window. That means you should have a statemenat that begins with `layout =`. Sorry to be stuck on this point, but so many people seem to have trouble following this simple instruction.

THE GOLDEN RULE OF WINDOW LAYOUTS

Thou shalt not re-use a windows's layout.... ever!

Or more explicitly put....

> If you are calling `Window` then you should define your window layout in the statement just prior to the `Window` call.

Demo Programs For Multiple Windows

There are several "Demo Programs" that will help you run multiple windows. Please download these programs and FOLLOW the example they have created for you.

Here is ***some*** of the code patterns you'll find when looking through the demo programs.

Multi-Window Design Pattern 1 - both windows active

```
import PySimpleGUI as sg



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

win1 = sg.Window('Window 1', layout)

win2_active = False
while True:
    ev1, vals1 = win1.Read(timeout=100)
    win1.FindElement('_OUTPUT_').Update(vals1[0])
    if ev1 is None or ev1 == 'Exit':
        break

    if not win2_active and ev1 == 'Launch 2':
        win2_active = True
        layout2 = [[sg.Text('Window 2')],
                   [sg.Button('Exit')]]

        win2 = sg.Window('Window 2', layout)

    if win2_active:
        ev2, vals2 = win2.Read(timeout=100)
        if ev2 is None or ev2 == 'Exit':
            win2_active  = False
            win2.Close()
```

Multi-Window Design Pattern 2 - only 1 active window

```
import PySimpleGUIQt as sg


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

win1 = sg.Window('Window 1', layout)
win2_active=False
while True:
    ev1, vals1 = win1.Read(timeout=100)
    if ev1 is None:
        break
    win1.FindElement('_OUTPUT_').Update(vals1[0])

    if ev1 == 'Launch 2'  and not win2_active:
        win2_active = True
        win1.Hide()
        layout2 = [[sg.Text('Window 2')],
                   [sg.Button('Exit')]]

        win2 = sg.Window('Window 2', layout)
        while True:
            ev2, vals2 = win2.Read()
            if ev2 is None or ev2 == 'Exit':
                win2.Close()
                win2_active = False
                win1.UnHide()
                break
```

The PySimpleGUI Debugger

Listen up if you are * advanced programmers debugging some really hairy stuff * programmers from another era that like to debug this way * those that want to have "x-ray vision" into their code * asked to use debugger to gather information * running on a platform that lacks ANY debugger * debugging a problem that happens only outside of a debugger environment * finding yourself saying "but it works when running PyCharm"

Starting on June 1, 2019, a built-in version of the debugger `imwatchingyou` has been shipping in every copy of PySimpleGUI. It's been largely downplayed to gauge whether or not the added code and the added feature and the use of a couple of keys, would mess up any users. Over 30,000 users have installed PySimpleGUI since then and there's not be a single Issue filed nor comment/complaint made, so seems safe enough to normal users... so far....

So far no one has reported anything at all about the debugger. The assumption is that it is quietly lying dormant, waiting for you to press the `BREAK` or `CONTROL` + `BREAK` keys. It's odd no one has accidently done this and freaked out, logging an Issue.

The plain PySimpleGUI module has a debugger builtin. For the other ports, please use the package `imwatchingyou`.

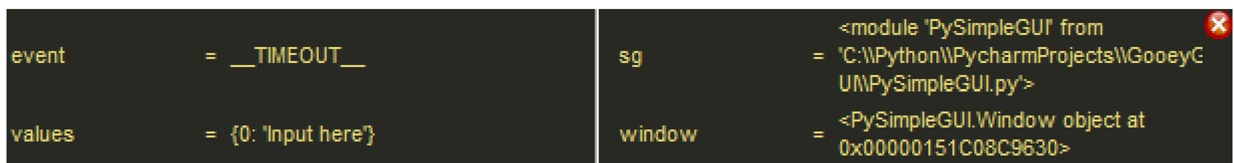
What is it? Why use it? What the heck? I already have an IDE.

This debugger provides you with something unique to most typical Python developers, the ability to "see" and interact with your code, **while it is running**. You can change variable values while your code continues to run.

Print statements are cool, but perhaps you're tired of seeing a printout of `event` and `values`:

```
Push Me {0: 'Input here'}
Push Me {0: 'Input here'}
Push Me {0: 'Input here'}
```

And would prefer to see this window updating continuously in the upper right corner of your display:



Notice how easy it is, using this window alone, to get the location that your PySimpleGUI package is coming from ***for sure***, no guessing. Expect this window to be in your debugging future as it'll get asked for from time to time.

Preparing To Run the Debugger

If your program is running with blocking `Read` calls, then you will want to add a timeout to your reads. This is because the debugger gets it's cycles by stealing a little bit of time from these async calls... but only when you have one of these debugger windows open so no bitching about wasted CPU time as there is none.

Your event loop will be modified from this blocking:

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

To this non-blocking:

```
while True:
    event, values = window.Read(timeout=200)
    if event == sg.TIMEOUT_KEY:
        continue
```

These 3 lines will in no way change how your application looks and performs. You can do this to any PySimpleGUI app that uses a blocking read and you'll not notice a difference. The reason this is a NOP (No-operation) is that when a timeout happens, the envent will be set to `sg.TIMEOUT_KEY`. If a timeout is returned as the event, the code simply ignores it and restarts the loop by executing a `continue` statement.

This timeout value of 200 means that your debugger GUI will be updated 5 times a second if nothing is happening. If this adds too much "drag" to your application, you can make the timeout larger. Try using 500 or 1000 instead of 100.

What happens if you don't add a timeout

Let's say you're in a situation where a very intermettent bug has just happened and the debugger would really help you, but you don't have a timeout on your `windows.Read()` call. It's OK. Recall that the way the debugger gets its "cycles" is to borrow from your `Read` calls. What you need to do is alternate between using the debugger and then generating another pass through your event loop.

Maybe it's an OK button that will cause your loop to execute again (without exiting). If so, you can use it to help move the debugger along.

Yes, this is a major pain in the ass, but it's not THAT bad and compared to nothing in a time of crisis and this is potentially your "savior tool" that's going to save your ass, pressing that OK button a few times is going to look like nothing to you. You just want to dump out the value of a variable that holds an instance of your class!

A Sample Program For Us To Use

Now that you understand how to add the debugger to your program, let's make a simple little program that you can use to follow these examples:

```
import PySimpleGUI as sg

window = sg.Window('Testing the Debugger', [[sg.Text('Debugger Tester'), sg.In('Input here'), sg.B('Push
Me')]])

while True:
    event, values = window.Read(timeout=500)
    if event == sg.TIMEOUT_KEY:
        continue
    if event is None:
        break
    print(event, values)
window.Close()
```

Debugger Windows

"Popout Debugger Window"

There are 2 debugger windows. One is called the "Popout" debugger window. The Popout window displays as many currently in-scope local variables as possible. This window is not interactive. It is meant to be a frequently updated "dashboard" or "snapshot" of your variables.

One "variable" shown in the popout window that is an often asked for piece of information when debugging Issues and that variable is `sg` (or whatever you named the PySimpleGUI pacakge when you did your import). The assumption is that your import is `import PySimpleGUI as sg`. If your import is different, then you'll see a different variable. The point is that it's shown here.

Exiting this window is done via the little red X, **or using the rickt-click menu** which is also used as one way to launch the Main Debugger Window

Ways of Launching the Popout Window

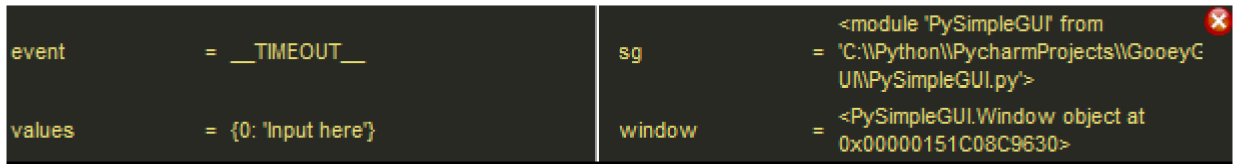
There are 3 ways of opening the Popout window.

1. Press the `BREAK` key on your keyboard.
2. Call the function `show_debugger_popout_window(location=(x,y))`
3. Add `Debug()` button to your layout - adds a little purple and yellow PySimpleGUI logo to your window

When you are asked for the "Location of your PySimpleGUI package or PySimpleGUI.py file" do this

If you wish to use the debugger to find the location of THIS running program's PySimpleGUI package / the PySimpleGUI.py file, then all you need to do is: * Press the `BREAK` key on your keyboard. * This is sometimes labelled as the `Cancel` key * May also have `Pause` printed on

key * On some US keyboards, it is located next to `Scroll Lock` and/or above `PageUp` key * This will open a window located in the upper right corner of your screen that looks something like this:



* The information you are seeking is shown next to the `sg` in the window You don't need to modify your program to get this info using this technique.

If your variable's value is too long and doesn't fit, then you'lll need to collect this information using the "Main Debugger Window"

What's NOT Listed In The Popout Debugger Window

The Popup window is a "Snapshot" of your local variables at the time the window was opened. This means **any variables that did not exist at the time the Popout was created will not be shown**. This window does **NOT** expand in size by adding new variables. Maybe in the future.

The "Main Debugger Window"

Now we're talking serious Python debugging!

Ever wish you had a `repl>>>` prompt that you could run while your program is running. Well, that's pretty much what you're getting with the PySimpleGUI debugger Main Window! Cool, huh? If you're not impressed, go get a cup of coffee and walk off that distraction in your head before carring on because we're in to some seriously cool shit here....

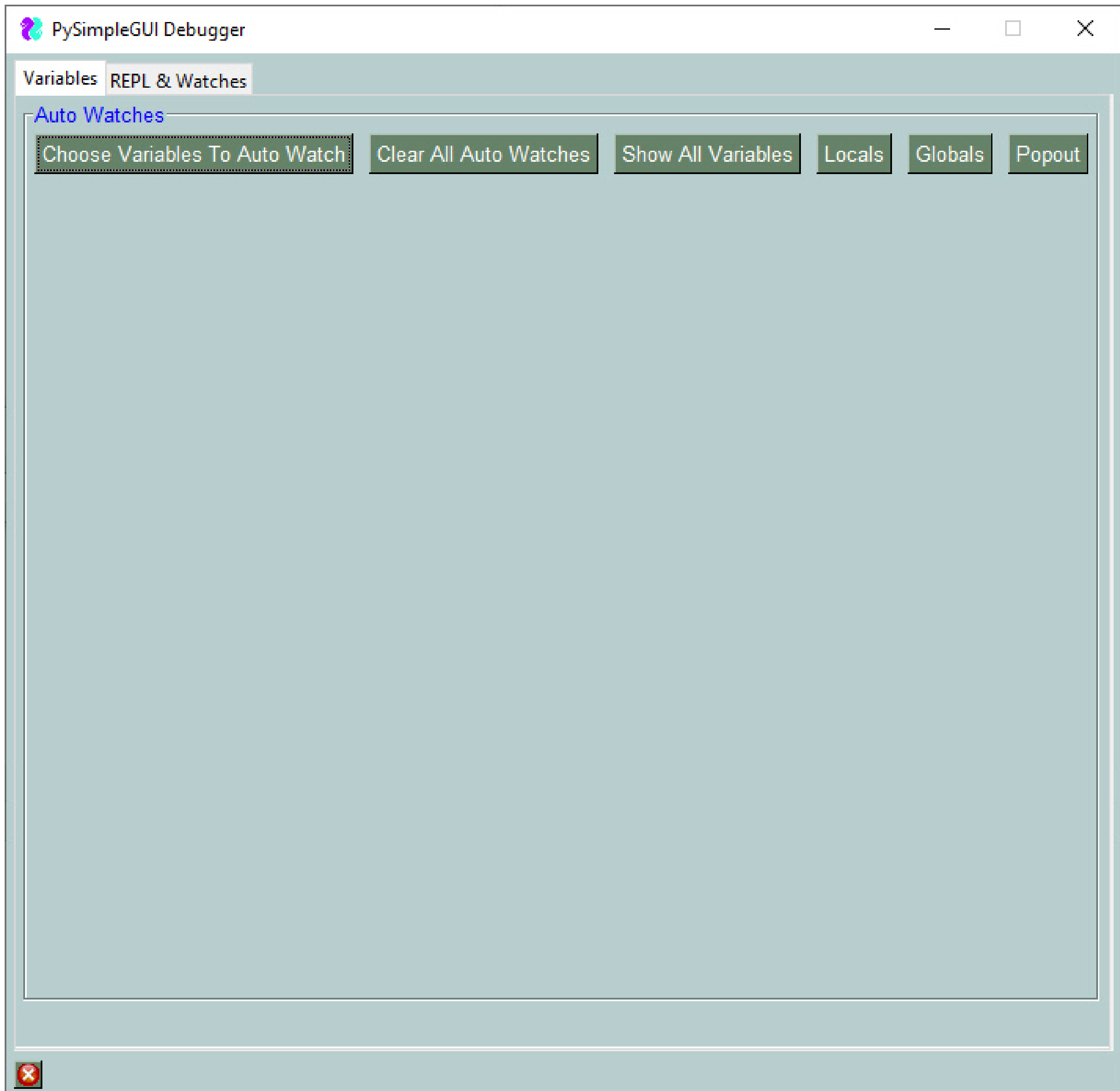
You'll find that this window has 2 tabs, one is labelled `Variables` and the other is labelled `REPL & Watches`

Ways of Opening the Main Debugger Window

There are 3 ways to open the Main Debugger Window

1. Press `Control` + `Break` on your PC keyboard
2. From the Popout Debug Window, right click and choose `Debugger` from the right click menu
3. From your code call `show_debugger_window(location=(x,y))`

The "Variables" Tab of Main Debugger Window

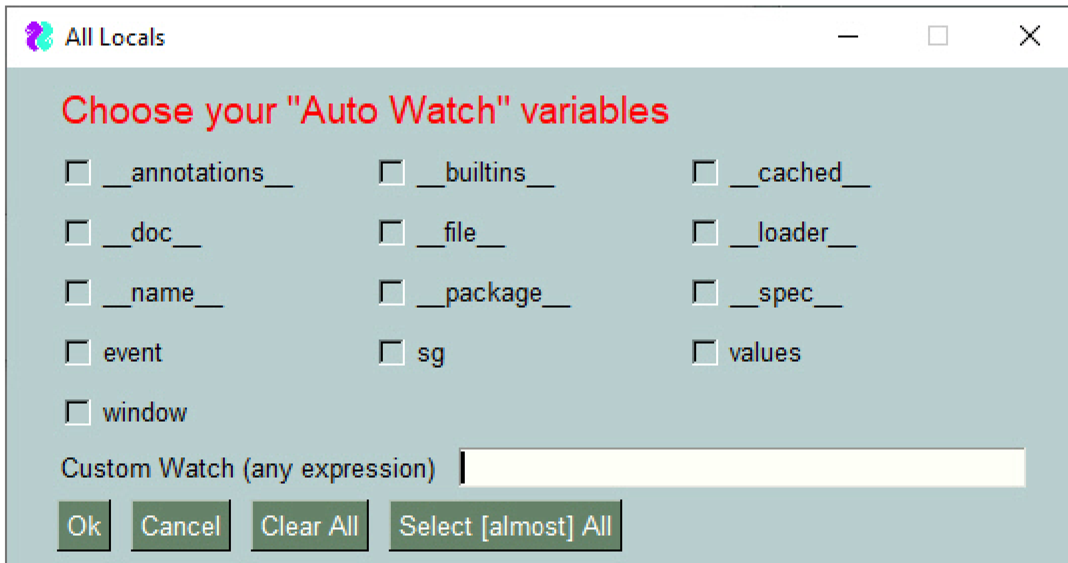


Notice the the "frame" surrounding this window is labelled "Auto Watches" in blue. Like the Popup window, this debugger window also "Watches" variables, which means continuously updates them as often as you call `Window.Read`.

The maximum number of "watches" you can have any any one time is 9.

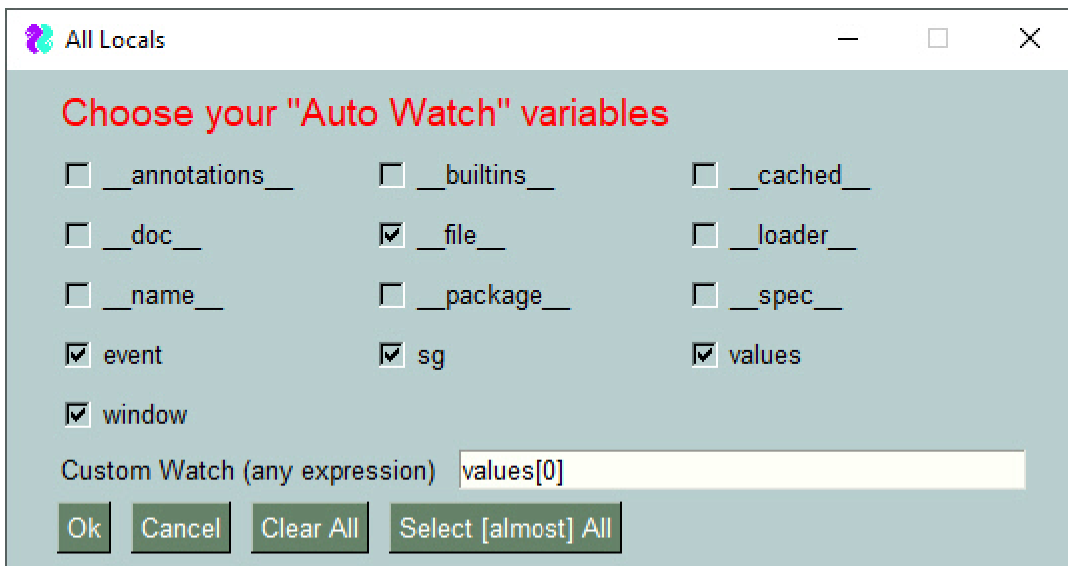
Choosing variables to watch

You can simply click "Show All Variable" button and the list of watched variables will be automatically populard by the first 9 variables it finds. Or you can click the "Choose Variables to Auto Watch" button where you can individually choose what variables, **and expressions** you wish to display.



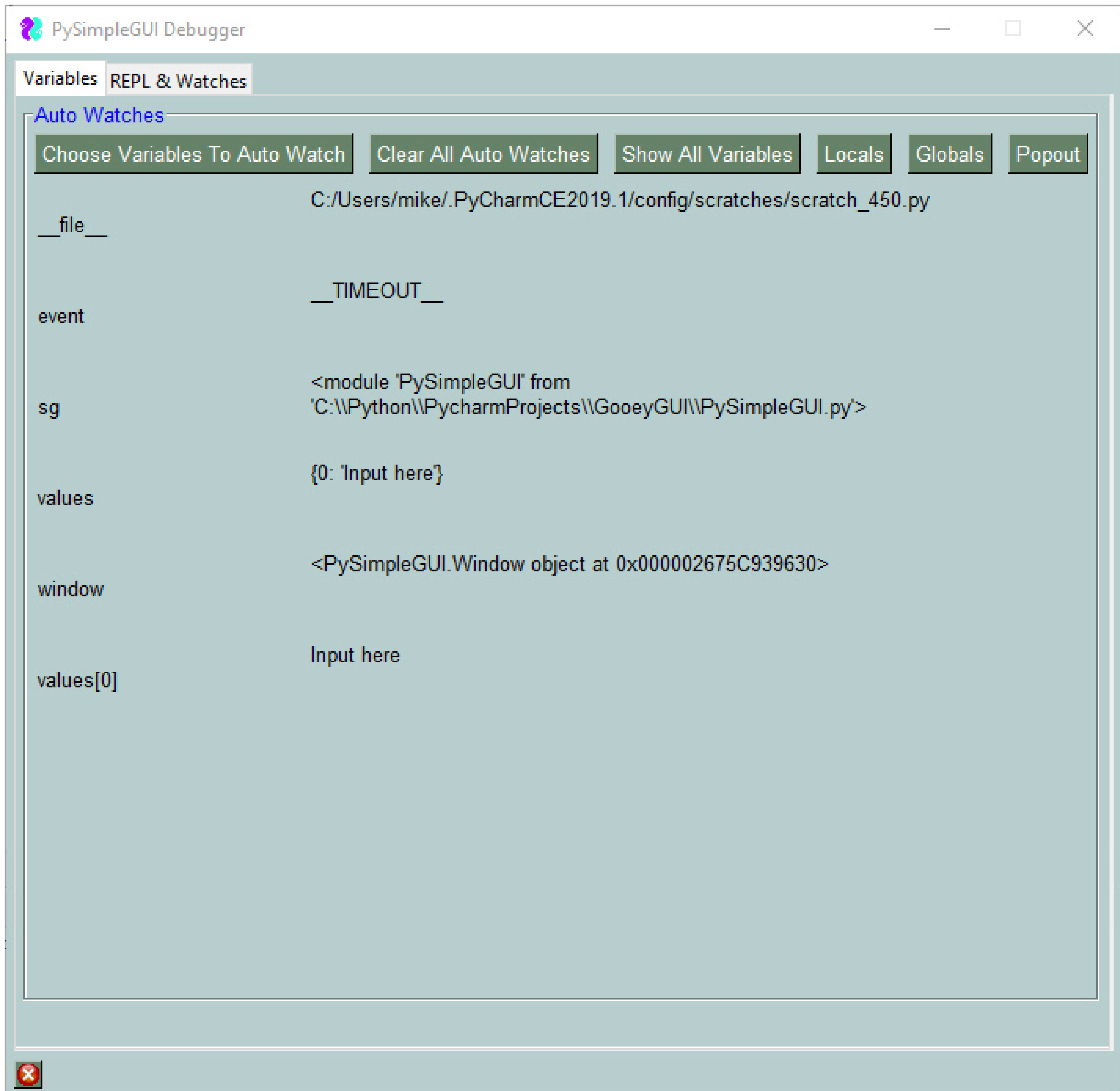
In this window we're checking checkboxes to display these variables:

`event` , `sg` , `values` , `window` , `__file__`



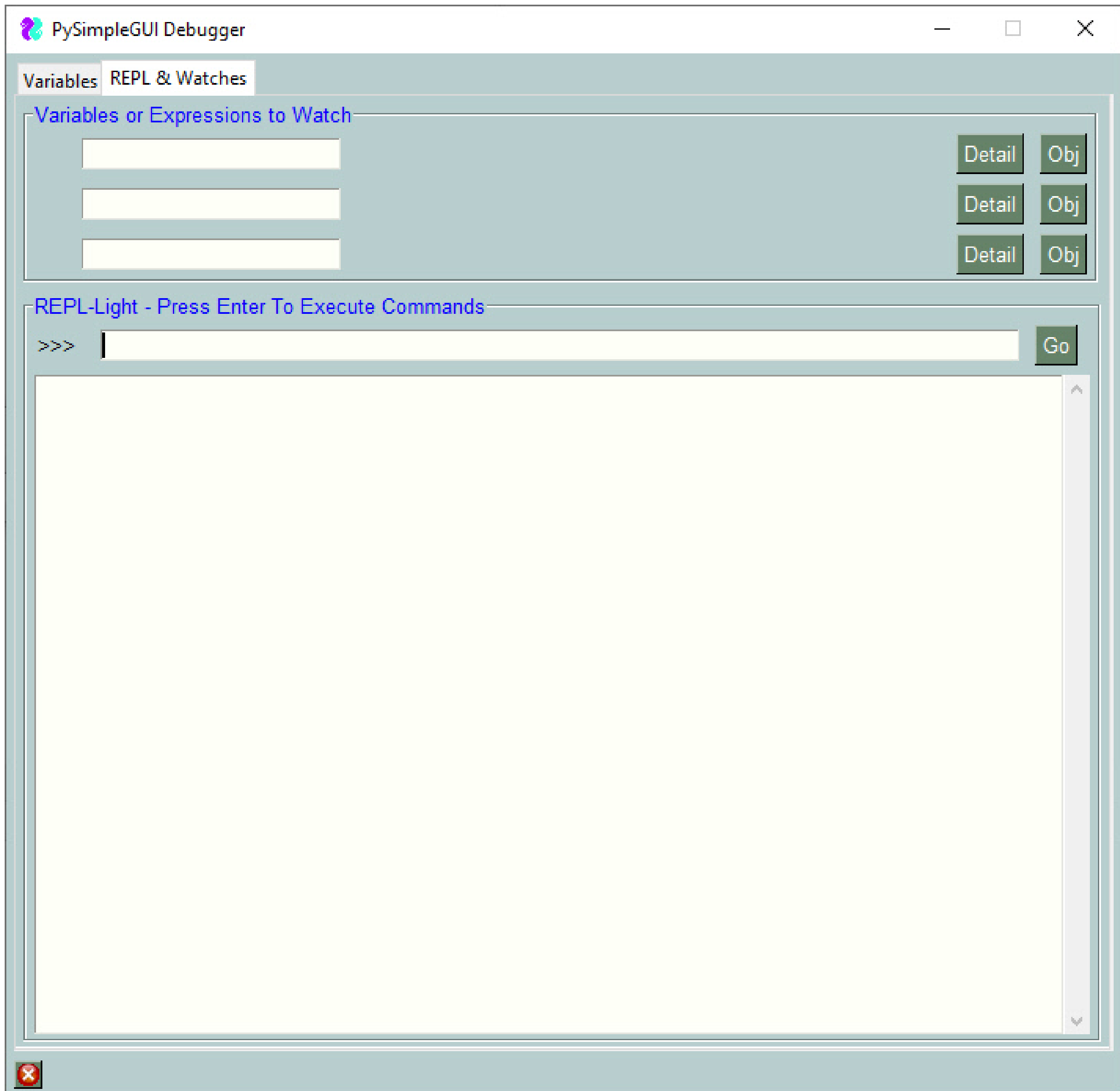
Additionally, you can see at the bottom of the window a "Custom Watch" has been defined. This can be any experession you want. Let's say you have a window with a LOT of values. Rather than looking through the `values` variable and finding the entry with the key you are looking for, the values variable's entry for a specific key is displayed.

In this example the Custom Watch entered was `values[0]` . After clicking on the "OK" button, indicating the variables are chosen that we wish to watch, this is the Main window that is shown:



We can see the variables we checked as well as the defined expression `values[0]` . If you leave this window open, these values with continuously be updated, on the fly, every time we call the line in our example code `window.Read(timeout=500)` . This means that the Main Debugger Window and these variables we defined will be updated every 500 milliseconds.

The REPL & Watches Tab



This tab is provided to you as a way to interact with your running program on a real-time basis.

If you want to quickly look at the values of variables, nearly ANY variables, then type the information into one of the 3 spaces provided to "Watch" either variables or experessions. In this example, the variable window was typed into the first slow.

Immediately after typing the character 'w', the information to the right was displayed. No button needs to be clicked. You merely neeed to type in a valid experession and it will be displayed to you.... and it will be displayed on an on-going, constantly-refreshing-basis.

PySimpleGUI Debugger

Variables REPL & Watches

Variables or Expressions to Watch

window <PySimpleGUI.Window object at 0x000002675C939630> Detail Obj

Detail Obj

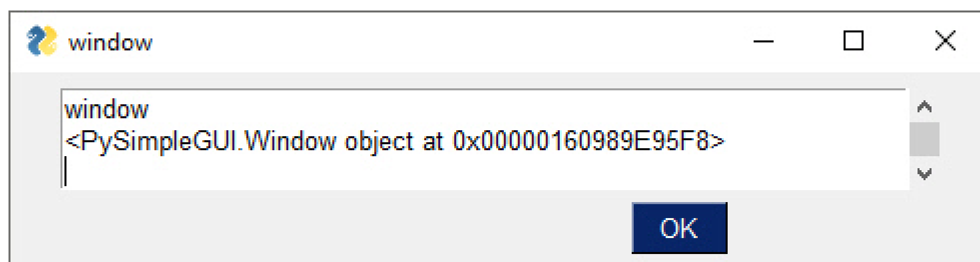
Detail Obj

REPL-Light - Press Enter To Execute Commands

>>> import matplotlib as m Go

If the area to the right of the input field is too small, then you can click on the "Detail" button and you will be shown a popup, scrolled window with all of the information displayed as if it were printed.

I'm sure you've had the lovely experience of printing an object. When clicking the "Detail" button next to the `window` variable being shown, this window is shown:



Oh, Python, -sigh-. I just want to see my `window` object printed.

`Obj` Button to the Rescue!

PySimpleGUI has a fun and very useful function that is discussed in the docs named `ObjToString` which takes an object and converts it's **contents** it into a nicely formatted string. This function is used to create the text output when you click the `Obj` button. The result is this instead of the tiny window shown previously:

window

```
window
<class 'PySimpleGUI.Window'>
'AllKeysDict' = {0: <PySimpleGUI.InputText object at
0x000002675C938048>, 'Push Me': <PySimpleGUI.Button object at
0x000002675C938080>}
'AutoClose' = False
'AutoCloseDuration' = 3
'AutoSizeButtons' = True
'AutoSizeText' = True
'BackgroundColor' = None
'BorderDepth' = None
'ButtonColor' = ('white', '#082567')
'ContainerElemementNumber' = 1
'CurrentlyRunningMainloop' = False
'DebuggerEnabled' = True
'DefaultButtonElementSize' = (10, 1)
'DefaultElementSize' = (45, 1)
'DictionaryKeyCounter' = 0
'DisableClose' = False
'DisableMinimize' = False
'ElementPadding' = (5, 3)
'Font' = ('Helvetica', 10)
'ForceTopLevel' = False
'FormRemainedOpen' = True
'GrabAnywhere' = False
'KeepOnTop' = False
'LastButtonClicked' = None
'LastButtonClickedWasRealtime' = False
'LastKeyboardEvent' = None
'Location' = (None, None)
'Margins' = (10, 5)
'NoTitleBar' = False
'NonBlocking' = False
'NumCols' = 3
'NumRows' = 1
'ParentWindow' = None
'ProgressBarColor' = (None, None)
'RadioDict' = {}
'Resizable' = False
'ReturnKeyboardEvents' = False
'ReturnValues' = ('__TIMEOUT__', {0: 'Input here'})
'ReturnValuesDictionary' = {0: 'Input here'}
'ReturnValuesList' = ['Input here']
'RightClickMenu' = None
'RootNeedsDestroying' = False
'Rows' = [[<PySimpleGUI.Text object at 0x000002675C8F1F60>,
<PySimpleGUI.InputText object at 0x000002675C938048>,
<PySimpleGUI.Button object at 0x000002675C938080>]]
'Shown' = True
'TKAfterID' = 'after#1089'
```

OK

The REPL Prompt

While not **really** a Python REPL prompt, this window's `REPL >>>` prompt is meant to act as much like one as possible. Here you can enter experessions and code too.

The uses for this prompt are so numerous and diverse that listing them all won't be attempted.

Your "XRay" and "Endoscope" into Your Program

Think of this prompt as a way to get specific diagnostics information about your ***running*** program. It cannot be stressed enough that the power and the usefullness of this tool is in its ability to diagnose a running program, after you've already started it running.

Execute Code

In addition to displaying information, getting paths to packages, finding version information, you can execute code from the PySimpleGUI Debugger's `REPL >>>` prompt. You can type in any expression as well as any **executable statement**.

For example, want to see what `PopupError` looks like while you're running your program. From the REPL prompt, type: `sg.PopupError('This is an error popup')`

The result is that you are shown a popup window with the text you supplied.

KNOW Answers to Questions About Your Program

Using this runtime tool, you can be confident in the data you collect. Right?

There's no better way to find what version of a package that your program is using than to ask your program. This is so true. Think about it. Rather than go into PyCharm, look at your project's "Virtual Environment", follow some path to get to a window that lists packages installed for that project, get the verstion and your're done, right? Well, maybe. But are you CERTAIN your program is using THAT version of the package in question?

SO MUCH time has been wasted in the past when people KNEW, for sure, what version they were running. Or, they had NO CLUE what version, or no clue to find out. There's nothing wrong with not knowing how to do something. We ALL start there. Geeez..

A real world example.....

How To Use the Debugger to Find The Version Number of a Package

Let's pull together everything we've learned to now and use the debugger to solve a problem that happens often and sometimes it's not at all obvious how to find the answer.

We're using ***Matplotlib*** and want to find the "Version".

For this example, the little 12-line program in the section "A Sample Program For Us To Use" is being used.

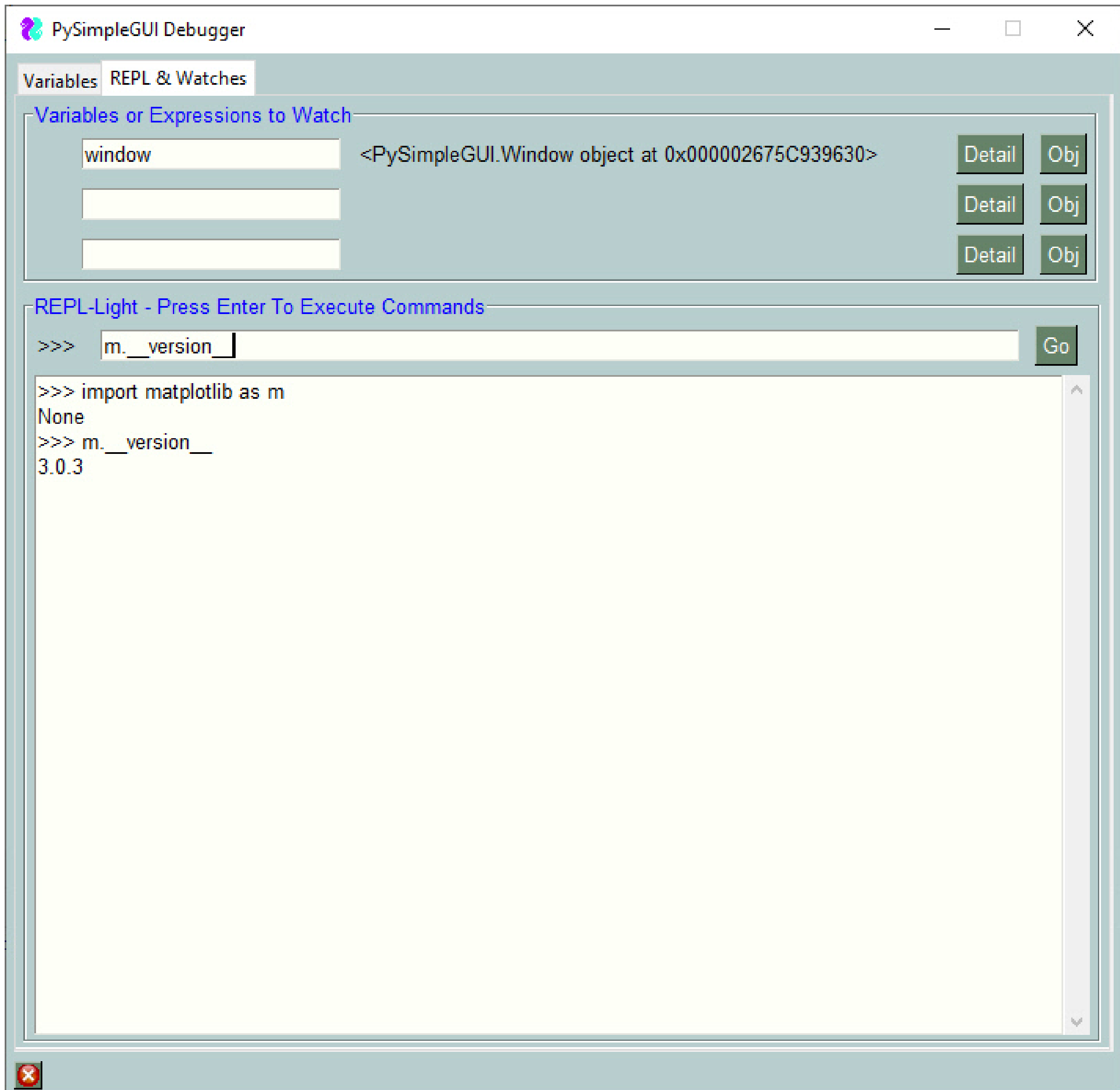
That program does not import `matplotlib`. We have a couple of choices, we can change the code, we can can import the package from the debugger. Let's use the debgger.

Pull up the Main Debugger Window by pressing `CONTROL+BREAK` keys. Then click the "REPL * Watches" tab. At the `>>>` prompt we'll first import the package by typing: `import matplotlib as m`

The result returned from Python calls that don't return anything is the value None. You will see the command you entered in the output area followed by "None", indicating success.

finally, type: `m.__version__`

The entire set of operations is shown in this window:



By convention you'll find many modules have a variable `__version__` that has the package's version number. PySimpleGUI has one. As you can see matplotlib has one. The `requests` module has this variable.

For maximum compatibility, PySimpleGUI not only uses `__version__`, but also has the version contained in another variable `version` which has the version number because in some situations the `__version__` is not available but the `version` variable is avaiable.

It is recommended that you use the variable `version` to get the PySimpleGUI version as it's so far been the most successful method.

tkinter, however does NOT.... of course.... follow this convention. No, to get the tkinter version, you need to look at the variable: `TkVersion`

Here's the output from the REPL in the debugger showing the tkinter version:

```
>>> import tkinter as t
None
>>> t.TkVersion
8.6
>>> t.__version__
Exception module 'tkinter' has no attribute '__version__'
```

Extending PySimpleGUI

PySimpleGUI doesn't and can't provide every single setting available in the underlying GUI framework. Not all tkinter options are available for a `Text` Element. Same with PySimpleGUIQt and the other ports.

There are a few of reasons for this.

1. Time & resource limits - The size of the PySimpleGUI development team is extremely small
2. PySimpleGUI provides a "Unified API". This means the code is, in theory, portable across all of the PySimpleGUI ports without chaning the user's code (except for the import)
3. PySimpleGUI is meant, by design, to be simple and cover 80% of the GUI problems.

However, PySimpleGUI programs are ***not*** dead ends!! Writing PySimpleGUI code and then getting to a point where you really really feel like you need to extend the Listbox to include the ability to change the "Selected" color. Maybe that's super-critical to your project. And maybe you find out late that the base PySimpleGUI code doesn't expose that tkinter capability. Fear not! The road does continue!!

Widget Access

Most of the user extensions / enhancements are at the "Element" level. You want some Element to do a trick that you cannot do using the existing PySimpleGUI APIs. It's just not possible. What to do?

What you need is access to the underlying GUI framework's "Widget". The good news is that you HAVE that access ready and waiting for you, for all of the ports of PySimpleGUI, not just the tkinter one.

`Element.Widget` is The GUI Widget

The class variable `Widget` contains the tkinter, Qt, WxPython, or Remi widget. With that variable you can modify that widget directly.

You must first `Read` or `Finalize` the window before accessing the `Widget` class variable

The reason for the Finalize requirement is that until a Window is Read or is Finalized it is not actually created and populated with GUI Widgets. The GUI Widgets are created when you do these 2 operations.

Side note - You can stop using the `.Finalize()` call added onto your window creation and instead use the `finalize` parameter in the `Window` call.

OLD WAY:

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

THE NEW WAY:

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

It's cleaner and less confusing for beginners who aren't necessarily trained in how chaining calls work. Py**Simple**GUI.

Example Use of `Element.Widget`

So far there have been 2 uses of this capability. One already mentioned is adding a new capability. The other way it's been used has been to fix a bug or make a workaround for a quirky behavior.

A recent Issue posted was that focus was always being set on a button in a tab when you switch tabs in tkinter. The user didn't want this to happen as it was putting an ugly black line around their nicely made graphical button.

There is no current way in PySimpleGUI to "disable focus" on an Element. That's essentially what was needed, the ability to tell tkinter that this widget should never get focus.

There is a way to tell tkinter that a widget should not get focus. The downside is that if you use your tab key to navigate, that element will never get focus. So, it's not only blocking focus for this automatic problem, but blocking it for all uses. Of course you can still click on the button.

The way through for this user was to modify the tkinter widget directly and tell it not to get focus. This was done in a single line of code:

```
window[button_key].Widget.config(takefocus=0)
```

The absolute beauty to this solution is that tkinter does NOT need to be imported into the user's program for this statement to run. Python already know what kind of object `.Widget` is and can thus show you the various methods and class variables for that object. Most all tkinter

options are strings so you don't need to import tkinter to get any enums.

Finding Your Element's Widget Type

Of course, in order to call the methods or access the object's class variables, you need to know the type of the underlying Widget being used. This document could list them all, but the downside is the widget could change types (not a good thing for people using the .Widget already!). It also saves space and time in getting this documentation published and available to you.

So, here's the way to get your element's widget's type:

```
print(type(window[your_element_key].Widget))
```

In the case of the button example above, what is printed is:

`<class 'tkinter.Button'>`

I don't think that could be any clearer. Your job at this point is to look at the tkinter documentation to see what the methods are for the tkinter `Button` widget.

Window Level Access

For this one you'll need some specific variables for the time being as there is no `Window` class variable that holds the window's representation in the GUI library being used.

For tkinter, at the moment, the window's root object is this:

```
sg.Window.TKroot
```

The type will vary in PySimpleGUI. It will either be: `tkinter.Tk()` `tkinter.Toplevel()`

Either way you'll access it using the same `Window` variable `sg.Window.TKroot`

Watch this space in the future for the more standardized variable name for this object. It may be something like `Window.Widget` as the Elements use or something like `Window.GUIWindow` .

Binding tkiner "events"

If you wish to receive events directly from tkinter, but do it in a PySimpleGUI way, then there's a particular way at the moment to make this happen.

tkinter performs a callback into user code when an event happens, but that's not how PySimpleGUI works. Instead of callbacks, a PySimpleGUI user's program simply returns an event via the `window.read()` call. In order for your "event" to generate an event that will be

returned to you via your read call, follow these instructions:

1. Create a Button for each event you wish to receive
2. Set visible=False when creating the buttons
3. Make the Button text be the event you want to see returned to you or set the button's Key to that value
4. After creating / finalizing the window, make the tkinter bind call, passing `element.ButtonReboundCallback` as the function to call.

This sample code binds not an element events but events from the window itself. In this case, Focus events.

```
import PySimpleGUI as sg

layout = [  [sg.Text('My Window')],
            [sg.Input(key='-IN-'), sg.Text('', key='-OUT-')],
            [sg.Button('Do Something'), sg.Button('Exit'),
             sg.Button('-FOCUS-IN-', visible=False), sg.Button('-FOCUS-OUT-', visible=False)]  ]

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

window.TKroot.bind("<FocusIn>", window['-FOCUS-IN-'].ButtonReboundCallback)
window.TKroot.bind("<FocusOut>", window['-FOCUS-OUT-'].ButtonReboundCallback)
```

This code binds the right mouse button to a button so that you can right click a button and get a different event than if you left clicked it.

```
import PySimpleGUI as sg

layout = [  [sg.Text('My Window')],
            [sg.Input(key='-IN-'), sg.Text('', key='-OUT-')],
            [sg.Button('Do Something'), sg.Button('Right Click Me')],
            [sg.Button('-RIGHT-', visible=False)]
            ]

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

window['Right Click Me'].Widget.bind("<Button-3>", window['-RIGHT-'].ButtonReboundCallback)

has_focus = True
while True:
    event, values = window.read()
    print(event, values)
    if event in (None, 'Exit'):
        break
window.close()
```

ELEMENT AND FUNCTION CALL REFERENCE

This reference section was previously intermixed with the text explanation, diagrams, code samples, etc. That was OK early on, but now that there are more Elements and more methods are being added on a fequent basis, it means that keeping this list updated is a difficult chore if it has a lot of text all around it.

Hoping this is a change for the better and that users will be able to find the information they seek quicker.

NOTE that this documentatiuopn section is created using the ***GitHUB released PySimpleGUI.py file***. Some of the calls may not be available to you or your port (Qt, Wx, Web). And some of the parameters may be different. We're working on adding docstrings to all the ports which will enable this kind of document to be available for each port.

Caution - Some functions / methods may be internal only yet exposed in this documenation

This section of the documentation is generated directly from the source code. As a result, sometimes internal only functions or methods that you are not supposed to be calling are accidently shown in this documentation. Hopefully these accidents don't happen often.

Without further delay... here are all of the Elements and the Window class

Button Element

Button Element - Defines all possible buttons. The shortcuts such as Submit, FileBrowse, ... each create a Button

```
Button(button_text="",
    button_type=7,
    target=(None, None),
    tooltip=None,
    file_types=(('ALL Files', '*.*'),),
    initial_folder=None,
    disabled=False,
    change_submits=False,
    enable_events=False,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	(str) Text to be displayed on the button
button_type	(int) You should NOT be setting this directly. ONLY the shortcut functions set this
target	Union[str, Tuple[int, int]] key or (row,col) target for the button. Note that -1 for column means 1 element to the left of this one. The constant ThisRow is used to indicate the current row. The Button itself is a valid target for some types of button
tooltip	(str) text, that will appear when mouse hovers over the element
file_types	Tuple[Tuple[str, str], ...] the filetypes that will be used to match files. To indicate all files: (("ALL Files", "."),). Note - NOT SUPPORTED ON MAC
initial_folder	(str) starting path for folders and files
disabled	(bool) If True button will be created disabled

Name	Meaning
click_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. If this button is a target, should it generate an event when filled in
image_filename	(str) image filename if there is a button image. GIFs and PNGs only.
image_data	Union[bytes, str] Raw or Base64 representation of the image to put on button. Choose either filename or data
image_size	Tuple[int, int] Size of the image in pixels (width, height)
image_subsample	(int) amount to reduce the size of the image. Divides the size by this number. 2=1/2, 3=1/3, 4=1/4, etc
border_width	(int) width of border around button in pixels
size	Tuple[int, int] (width, height) of the button in characters wide, rows high
auto_size_button	(bool) if True the button size is sized to fit the text
button_color	Tuple[str, str] (text color, background color) of button. Easy to remember which is which if you say "ON" between colors. "red" on "green". Note - Does not always work on Macs
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
bind_return_key	(bool) If True the return key will cause this button to be pressed
focus	(bool) if True, initial focus will be put on this button
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonCallBack

Not user callable! Called by tkinter when a button is clicked. This is where all the fun begins!

```
ButtonCallBack()
```

ButtonPressCallBack

Not a user callable method. Callback called by tkinter when a "realtime" button is pressed

```
ButtonPressCallBack(parm)
```

Parameter Descriptions:

Name	Meaning
parm	Event info passed in by tkinter

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

ButtonReleaseCallBack

Not a user callable function. Called by tkinter when a "realtime" button is released

```
ButtonReleaseCallBack(parm)
```

Parameter Descriptions:

Name	Meaning

Name	Meaning
parm	the event info from tkinter

Click

Generates a click of the button as if the user clicked the button Calls the tkinter invoke method for the button

```
Click()
```

GetText

Returns the current text shown on a button

`GetText()`

Name	Meaning
return	(str) The text currently displayed on the button

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Button Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(text=None,
    button_color=(None, None),
    disabled=None,
    image_data=None,
    image_filename=None,
    visible=None,
    image_subsample=None,
    image_size=None)
```

Parameter Descriptions:

Name	Meaning
text	(str) sets button text
button_color	Tuple[str, str] (text color, background color) of button. Easy to remember which is which if you say "ON" between colors. "red" on "green"
disabled	(bool) disable or enable state of the element
image_data	Union[bytes, str] Raw or Base64 representation of the image to put on button. Choose either filename or data
image_filename	(str) image filename if there is a button image. GIFs and PNGs only.
visible	(bool) control visibility of element
image_subsample	(int) amount to reduce the size of the image. Divides the size by this number. 2=1/2, 3=1/3, 4=1/4, etc
image_size	Tuple[int, int] Size of the image in pixels (width, height)

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

click

Generates a click of the button as if the user clicked the button Calls the tkinter invoke method for the button

```
click()
```

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

get_text

Returns the current text shown on a button

`get_text()`

Name	Meaning
return	(str) The text currently displayed on the button

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Button Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(text=None,
    button_color=(None, None),
    disabled=None,
    image_data=None,
    image_filename=None,
    visible=None,
    image_subsample=None,
    image_size=None)
```

Parameter Descriptions:

Name	Meaning
text	(str) sets button text

Name	Meaning
button_color	Tuple[str, str] (text color, background color) of button. Easy to remember which is which if you say "ON" between colors. "red" on "green"
disabled	(bool) disable or enable state of the element
image_data	Union[bytes, str] Raw or Base64 representation of the image to put on button. Choose either filename or data
image_filename	(str) image filename if there is a button image. GIFs and PNGs only.
visible	(bool) control visibility of element
image_subsample	(int) amount to reduce the size of the image. Divides the size by this number. 2=1/2, 3=1/3, 4=1/4, etc
image_size	Tuple[int, int] Size of the image in pixels (width, height)

The Button Menu Element. Creates a button that when clicked will show a menu similar to right click menu

```
ButtonMenu(button_text,
    menu_def,
    tooltip=None,
    disabled=False,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    pad=None,
    key=None,
    tearoff=False,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	(str) Text to be displayed on the button

Name	Meaning
menu_def	List[List[str]] A list of lists of Menu items to show when this element is clicked. See docs for format as they are the same for all menu types
tooltip	(str) text, that will appear when mouse hovers over the element
disabled	(bool) If True button will be created disabled
image_filename	(str) image filename if there is a button image. GIFs and PNGs only.
image_data	Union[bytes, str] Raw or Base64 representation of the image to put on button. Choose either filename or data
image_size	Tuple[int, int] Size of the image in pixels (width, height)
image_subsample	(int) amount to reduce the size of the image. Divides the size by this number. 2=1/2, 3=1/3, 4=1/4, etc
border_width	(int) width of border around button in pixels
size	Tuple[int, int] (width, height) of the button in characters wide, rows high
auto_size_button	(bool) if True the button size is sized to fit the text
button_color	Tuple[str, str] (text color, background color) of button. Easy to remember which is which if you say "ON" between colors. "red" on "green"
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
tearoff	(bool) Determines if menus should allow them to be torn off
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Click

Generates a click of the button as if the user clicked the button Calls the tkinter invoke method for the button

```
Click()
```

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the ButtonMenu Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(menu_definition, visible=None)
```

Parameter Descriptions:

Name	Meaning
menu_definition	(List[List]) New menu definition (in menu definition format)
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the ButtonMenu Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(menu_definition, visible=None)
```

Parameter Descriptions:

Name	Meaning

Name	Meaning
menu_definition	(List[List]) New menu definition (in menu definition format)
visible	(bool) control visibility of element

Canvas Element

```
Canvas(canvas=None,
    background_color=None,
    size=(None, None),
    pad=None,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
canvas	(tk.Canvas) Your own tk.Canvas if you already created it. Leave blank to create a Canvas
background_color	(str) color of background
size	Tuple[int,int] (width in char, height in rows) size in pixels to make canvas
pad	Amount of padding to put around element
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

TKCanvas

property: TKCanvas

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

tk_canvas

property: tk_canvas

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

Checkbox Element

```
Checkbox Element - Displays a checkbox and text next to it

Checkbox(text,
    default=False,
    size=(None, None),
    auto_size_text=None,
    font=None,
    background_color=None,
    text_color=None,
    change_submits=False,
    enable_events=False,
    disabled=False,
    key=None,
    pad=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
text	(str) Text to display next to checkbox

Name	Meaning
default	(bool). Set to True if you want this checkbox initially checked
size	Tuple[int, int] (width, height) width = characters-wide, height = rows-high
auto_size_text	(bool) if True will size the element to match the length of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
background_color	(str) color of background
text_color	(str) color of the text
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Checkbox events happen when an item changes
disabled	(bool) set disable state
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text, that will appear when mouse hovers over the element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Return the current state of this checkbox

```
Get()
```

Name	Meaning
return	(bool) Current state of checkbox

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Checkbox Element. Must call `Window.Read` or `Window.Finalize` prior. Note that changing visibility may cause element to change locations when made visible after invisible

```
Update(value=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(bool) if True checks the checkbox, False clears it
disabled	(bool) disable or enable element
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

Return the current state of this checkbox

`get()`

Name	Meaning
return	(bool) Current state of checkbox

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Checkbox Element. Must call `Window.Read` or `Window.Finalize` prior. Note that changing visibility may cause element to change locations when made visible after invisible

```
update(value=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(bool) if True checks the checkbox, False clears it
disabled	(bool) disable or enable element
visible	(bool) control visibility of element

Column Element

A container element that is used to create a layout within your window's layout

```
Column(layout,
    background_color=None,
    size=(None, None),
    pad=None,
    scrollable=False,
    vertical_scroll_only=False,
    right_click_menu=None,
    key=None,
    visible=True,
    justification="left",
    element_justification="left",
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
layout	List[List[Element]] Layout that will be shown in the Column container
background_color	(str) color of background of entire Column

Name	Meaning
size	Tuple[int, int] (width, height) size in pixels (doesn't work quite right, sometimes only 1 dimension is set by tkinter
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
scrollable	(bool) if True then scrollbars will be added to the column
vertical_scroll_only	(bool) if Truen then no horizontal scrollbar will be shown
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
visible	(bool) set visibility state of the element
justification	(str) set justification for the Column itself. Note entire row containing the Column will be affected
element_justification	(str) All elements inside the Column will have this justification 'left', 'right', 'center' are valid values
metadata	(Any) User metadata that can be set to ANYTHING

AddRow

Not recommended user call. Used to add rows of Elements to the Column Element.

```
AddRow(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Layout

Can use like the Window.Layout method, but it's better to use the layout parameter when creating

```
Layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The rows of Elements
return	(Column) Used for chaining

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Column Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(visible=None)
```

Parameter Descriptions:

Name	Meaning
visible	(bool) control visibility of element

add_row

Not recommended user call. Used to add rows of Elements to the Column Element.

```
add_row(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback

function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

layout

Can use like the Window.Layout method, but it's better to use the layout parameter when creating

```
layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The rows of Elements
return	(Column) Used for chaining

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Column Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(visible=None)
```

Parameter Descriptions:

Name	Meaning
visible	(bool) control visibility of element

Combo Element

ComboBox Element - A combination of a single-line input and a drop-down menu. User can type in their own value or choose from list.

```
Combo(values,
    default_value=None,
    size=(None, None),
    auto_size_text=None,
    background_color=None,
    text_color=None,
    change_submits=False,
    enable_events=False,
    disabled=False,
    key=None,
    pad=None,
    tooltip=None,
    readonly=False,
    font=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] values to choose. While displayed as text, the items returned are what the caller supplied, not text
default_value	(Any) Choice to be displayed as initial value. Must match one of values variable contents
size	Tuple[int, int] (width, height) width = characters-wide, height = rows-high
auto_size_text	(bool) True if element should be the same size as the contents
background_color	(str) color of background
text_color	(str) color of the text
change_submits	(bool) DEPRICATED DO NOT USE. Use `enable_events` instead
enable_events	(bool) Turns on the element specific events. Combo event is when a choice is made
disabled	(bool) set disable state for element
key	(Any) Used with window.FindElement and with return values to uniquely identify this element

Name	Meaning
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text that will appear when mouse hovers over this element
readonly	(bool) make element readonly (user can't change). True means user cannot change
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Returns the current (right now) value of the Combo. DO NOT USE THIS AS THE NORMAL WAY OF READING A COMBO! You should be using values from your call to window.Read instead. Know what you're doing if you use it.

`Get()`

Name	Meaning
return	Union[Any, None] Returns the value of what is currently chosen

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Combo Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    values=None,
    set_to_index=None,
    disabled=None,
    readonly=None,
    font=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning

Name	Meaning
value	(Any) change which value is current selected hased on new list of previous list of choices
values	List[Any] change list of choices
set_to_index	(int) change selection to a particular choice starting with index = 0
disabled	(bool) disable or enable state of the element
readonly	(bool) if True make element readonly (user cannot change any choices)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

Returns the current (right now) value of the Combo. DO NOT USE THIS AS THE NORMAL WAY OF READING A COMBO! You should be using values from your call to window.Read instead. Know what you're doing if you use it.

`get()`

Name	Meaning
return	Union[Any, None] Returns the value of what is currently chosen

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Combo Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    values=None,
    set_to_index=None,
    disabled=None,
    readonly=None,
    font=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) change which value is current selected hased on new list of previous list of choices
values	List[Any] change list of choices
set_to_index	(int) change selection to a particular choice starting with index = 0
disabled	(bool) disable or enable state of the element
readonly	(bool) if True make element readonly (user cannot change any choices)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) control visibility of element

Frame Element

A Frame Element that contains other Elements. Encloses with a line around elements and a text label.

```
Frame(title,
    layout,
    title_color=None,
    background_color=None,
    title_location=None,
    relief="groove",
    size=(None, None),
    font=None,
    pad=None,
    border_width=None,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    element_justification="left",
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
title	(str) text that is displayed as the Frame's "label" or title
layout	List[List[Elements]] The layout to put inside the Frame
title_color	(str) color of the title text
background_color	(str) background color of the Frame
title_location	(enum) location to place the text title. Choices include: TITLE_LOCATION_TOP TITLE_LOCATION_BOTTOM TITLE_LOCATION_LEFT TITLE_LOCATION_RIGHT TITLE_LOCATION_TOP_LEFT TITLE_LOCATION_TOP_RIGHT TITLE_LOCATION_BOTTOM_LEFT TITLE_LOCATION_BOTTOM_RIGHT
relief	(enum) relief style. Values are same as other elements with reliefs. Choices include RELIEF_RAISED RELIEF_SUNKEN RELIEF_FLAT RELIEF_RIDGE RELIEF_GROOVE RELIEF_SOLID
size	Tuple[int, int] (width in characters, height in rows) (note this parameter may not always work)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))

Name	Meaning
border_width	(int) width of border around element in pixels
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
element_justification	(str) All elements inside the Frame will have this justification 'left', 'right', 'center' are valid values
metadata	(Any) User metadata that can be set to ANYTHING

AddRow

Not recommended user call. Used to add rows of Elements to the Frame Element.

```
AddRow(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Layout

Can use like the Window.Layout method, but it's better to use the layout parameter when creating

```
Layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The rows of Elements
return	(Frame) Used for chaining

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Frame Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) New text value to show on frame
visible	(bool) control visibility of element

add_row

Not recommended user call. Used to add rows of Elements to the Frame Element.

```
add_row(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

layout

Can use like the Window.Layout method, but it's better to use the layout parameter when creating

```
layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The rows of Elements
return	(Frame) Used for chaining

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Frame Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) New text value to show on frame
visible	(bool) control visibility of element

Graph Element

```
Creates an area for you to draw on.  The MAGICAL property this Element has is that you interact
with the element using your own coordinate system.  This is an important point!!  YOU define where the location
is for (0,0).  Want (0,0) to be in the middle of the graph like a math 4-quadrant graph?  No problem!  Set your
lower left corner to be (-100,-100) and your upper right to be (100,100) and you've got yourself a graph with
(0,0) at the center.
One of THE coolest of the Elements.
You can also use float values. To do so, be sure and set the float_values parameter.
Mouse click and drag events are possible and return the (x,y) coordinates of the mouse
Drawing primitives return an "id" that is referenced when you want to operation on that item (e.g. to erase it)
```

```
Graph(canvas_size,
    graph_bottom_left,
    graph_top_right,
    background_color=None,
    pad=None,
    change_submits=False,
    drag_submits=False,
    enable_events=False,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    float_values=False,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
canvas_size	Tuple[int, int] (width, height) size of the canvas area in pixels
graph_bottom_left	Tuple[int, int] (x,y) The bottoms left corner of your coordinate system
graph_top_right	Tuple[int, int] (x,y) The top right corner of your coordinate system
background_color	(str) background color of the drawing area
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
change_submits	(bool) * DEPRICATED DO NOT USE! Same as enable_events
drag_submits	(bool) if True and Events are enabled for the Graph, will report Events any time the mouse moves while button down
enable_events	(bool) If True then clicks on the Graph are immediately reported as an event. Use this instead of change_submits
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.

Name	Meaning
visible	(bool) set visibility state of the element (Default = True)
float_values	(bool) If True x,y coordinates are returned as floats, not ints
metadata	(Any) User metadata that can be set to ANYTHING

BringFigureToFront

Changes Z-order of figures on the Graph. Brings the indicated figure to the front of all other drawn figures

```
BringFigureToFront(figure)
```

Parameter Descriptions:

Name	Meaning
figure	(int) value returned by tkinter when creating the figure / drawing

ButtonPressCallBack

Not a user callable method. Used to get Graph click events. Called by tkinter when button is released

```
ButtonPressCallBack(event)
```

Parameter Descriptions:

Name	Meaning
event	(event) event info from tkinter. Contains the x and y coordinates of a click

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

ButtonReleaseCallBack

Not a user callable method. Used to get Graph click events. Called by tkinter when button is released

```
ButtonReleaseCallBack(event)
```

Parameter Descriptions:

Name	Meaning
event	(event) event info from tkinter. Note not used in this method

DeleteFigure

Remove from the Graph the figure represented by id. The id is given to you anytime you call a drawing primitive

```
DeleteFigure(id)
```

Parameter Descriptions:

Name	Meaning
id	(int) the id returned to you when calling one of the drawing methods

DrawArc

Draws different types of arcs. Uses a "bounding box" to define location

```
DrawArc(top_left,
    bottom_right,
    extent,
    start_angle,
    style=None,
    arc_color="black")
```

Parameter Descriptions:

Name	Meaning
top_left	Union[Tuple[int, int], Tuple[float, float]] the top left point of bounding rectangle
bottom_right	Union[Tuple[int, int], Tuple[float, float]] the bottom right point of bounding rectangle
extent	(float) Andle to end drawing. Used in conjunction with start_angle
start_angle	(float) Angle to begin drawing. Used in conjunction with extent
style	(str) Valid choices are One of these Style strings- 'pieslice', 'chord', 'arc', 'first', 'last', 'butt', 'projecting', 'round', 'bevel', 'miter'
arc_color	(str) color to draw arc with
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the arc

DrawCircle

Draws a circle, cenetered at the location provided. Can set the fill and outline colors

```
DrawCircle(center_location,
    radius,
    fill_color=None,
    line_color="black")
```

Parameter Descriptions:

Name	Meaning
center_location	Union [Tuple[int, int], Tuple[float, float]] Center location using USER'S coordinate system
radius	Union[int, float] Radius in user's coordinate values.
fill_color	(str) color of the point to draw
line_color	(str) color of the outer line that goes around the circle (sorry, can't set thickness)

Name	Meaning
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the circle

DrawImage

Places an image onto your canvas. It's a really important method for this element as it enables so much

```
DrawImage(filename=None,
    data=None,
    location=(None, None),
    color="black",
    font=None,
    angle=0)
```

Parameter Descriptions:

Name	Meaning
filename	(str) if image is in a file, path and filename for the image. (GIF and PNG only!)
data	Union[str, bytes] if image is in Base64 format or raw? format then use instead of filename
location	Union[Tuple[int, int], Tuple[float, float]] the (x,y) location to place image's top left corner
color	(str) text color
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
angle	(float) Angle 0 to 360 to draw the text. Zero represents horizontal text
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the image

DrawLine

Draws a line from one point to another point using USER'S coordinates. Can set the color and width of line

```
DrawLine(point_from,
    point_to,
    color="black",
    width=1)
```

Parameter Descriptions:

Name	Meaning
point_from	Union[Tuple[int, int], Tuple[float, float]] Starting point for line
point_to	Union[Tuple[int, int], Tuple[float, float]] Ending point for line
color	(str) Color of the line
width	(int) width of line in pixels
return	Union[int, None] id returned from tktiner or None if user closed the window. id is used when you want to manipulate the line

DrawOval

Draws an oval based on coordinates in user coordinate system. Provide the location of a "bounding rectangle"

```
DrawOval(top_left,
    bottom_right,
    fill_color=None,
    line_color=None)
```

Parameter Descriptions:

Name	Meaning
top_left	Union[Tuple[int, int], Tuple[float, float]] the top left point of bounding rectangle
bottom_right	Union[Tuple[int, int], Tuple[float, float]] the bottom right point of bounding rectangle
fill_color	(str) color of the interrior
line_color	(str) color of outline of oval

Name	Meaning
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the oval

DrawPoint

Draws a "dot" at the point you specify using the USER'S coordinate system

```
DrawPoint(point,
    size=2,
    color="black")
```

Parameter Descriptions:

Name	Meaning
point	Union [Tuple[int, int], Tuple[float, float]] Center location using USER'S coordinate system
size	Union[int, float] Radius? (Or is it the diameter?) in user's coordinate values.
color	(str) color of the point to draw
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the point

DrawRectangle

Draw a rectangle given 2 points. Can control the line and fill colors

```
DrawRectangle(top_left,
    bottom_right,
    fill_color=None,
    line_color=None,
    line_width=None)
```

Parameter Descriptions:

Name	Meaning
top_left	Union[Tuple[int, int], Tuple[float, float]] the top left point of rectangle

Name	Meaning
bottom_right	Union[Tuple[int, int], Tuple[float, float]] the bottom right point of rectangle
fill_color	(str) color of the interior
line_color	(str) color of outline
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the rectangle

DrawText

Draw some text on your graph. This is how you label graph number lines for example

```
DrawText(text,
    location,
    color="black",
    font=None,
    angle=0,
    text_location="center")
```

Parameter Descriptions:

Name	Meaning
text	(str) text to display
location	Union[Tuple[int, int], Tuple[float, float]] location to place first letter
color	(str) text color
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
angle	(float) Angle 0 to 360 to draw the text. Zero represents horizontal text
text_location	(enum) "anchor" location for the text. Values start with TEXT_LOCATION_
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the text

Erase

Erase the Graph - Removes all figures previously "drawn" using the Graph methods (e.g. DrawText)

```
Erase()
```

MotionCallBack

Not a user callable method. Used to get Graph mouse motion events. Called by tkinter when mouse moved

```
MotionCallBack(event)
```

Parameter Descriptions:

Name	Meaning
event	(event) event info from tkinter. Contains the x and y coordinates of a mouse

Move

Moves the entire drawing area (the canvas) by some delta from the current position. Units are indicated in your coordinate system indicated number of ticks in your coordinate system

```
Move(x_direction, y_direction)
```

Parameter Descriptions:

Name	Meaning
x_direction	Union[int, float] how far to move in the "X" direction in your coordinates
y_direction	Union[int, float] how far to move in the "Y" direction in your coordinates

MoveFigure

Moves a previously drawn figure using a "delta" from current position

```
MoveFigure(figure,
    x_direction,
    y_direction)
```

Parameter Descriptions:

Name	Meaning
figure	(id) Previously obtained figure-id. These are returned from all Draw methods
x_direction	Union[int, float] delta to apply to position in the X direction
y_direction	Union[int, float] delta to apply to position in the Y direction

RelocateFigure

Move a previously made figure to an arbitrary (x,y) location. This differs from the Move methods because it uses absolute coordinates versus relative for Move

```
RelocateFigure(figure,
    x,
    y)
```

Parameter Descriptions:

Name	Meaning
figure	(id) Previously obtained figure-id. These are returned from all Draw methods
x	Union[int, float] location on X axis (in user coords) to move the upper left corner of the figure
y	Union[int, float] location on Y axis (in user coords) to move the upper left corner of the figure

SendFigureToBack

Changes Z-order of figures on the Graph. Sends the indicated figure to the back of all other drawn figures

```
SendFigureToBack(figure)
```

Parameter Descriptions:

Name	Meaning
figure	(int) value returned by tkinter when creating the figure / drawing

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

TKCanvas

property: TKCanvas

Update

Changes some of the settings for the Graph Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(background_color=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
background_color	color of background

Name	Meaning
visible	(bool) control visibility of element

bring_figure_to_front

Changes Z-order of figures on the Graph. Brings the indicated figure to the front of all other drawn figures

```
bring_figure_to_front(figure)
```

Parameter Descriptions:

Name	Meaning
figure	(int) value returned by tkinter when creating the figure / drawing

button_press_call_back

Not a user callable method. Used to get Graph click events. Called by tkinter when button is released

```
button_press_call_back(event)
```

Parameter Descriptions:

Name	Meaning
event	(event) event info from tkinter. Contains the x and y coordinates of a click

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

button_release_call_back

Not a user callable method. Used to get Graph click events. Called by tkinter when button is released

```
button_release_call_back(event)
```

Parameter Descriptions:

Name	Meaning
event	(event) event info from tkinter. Note not used in this method

delete_figure

Remove from the Graph the figure represented by id. The id is given to you anytime you call a drawing primitive

```
delete_figure(id)
```

Parameter Descriptions:

Name	Meaning
id	(int) the id returned to you when calling one of the drawing methods

draw_arc

Draws different types of arcs. Uses a "bounding box" to define location

```
draw_arc(top_left,
    bottom_right,
    extent,
    start_angle,
    style=None,
    arc_color="black")
```

Parameter Descriptions:

Name	Meaning
top_left	Union[Tuple[int, int], Tuple[float, float]] the top left point of bounding rectangle
bottom_right	Union[Tuple[int, int], Tuple[float, float]] the bottom right point of bounding rectangle
extent	(float) Andle to end drawing. Used in conjunction with start_angle
start_angle	(float) Angle to begin drawing. Used in conjunction with extent
style	(str) Valid choices are One of these Style strings- 'pieslice', 'chord', 'arc', 'first', 'last', 'butt', 'projecting', 'round', 'bevel', 'miter'
arc_color	(str) color to draw arc with
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the arc

draw_circle

Draws a circle, cenetered at the location provided. Can set the fill and outline colors

```
draw_circle(center_location,
    radius,
    fill_color=None,
    line_color="black")
```

Parameter Descriptions:

Name	Meaning
center_location	Union [Tuple[int, int], Tuple[float, float]] Center location using USER'S coordinate system
radius	Union[int, float] Radius in user's coordinate values.
fill_color	(str) color of the point to draw
line_color	(str) color of the outer line that goes around the circle (sorry, can't set thickness)

Name	Meaning
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the circle

draw_image

Places an image onto your canvas. It's a really important method for this element as it enables so much

```
draw_image(filename=None,
    data=None,
    location=(None, None),
    color="black",
    font=None,
    angle=0)
```

Parameter Descriptions:

Name	Meaning
filename	(str) if image is in a file, path and filename for the image. (GIF and PNG only!)
data	Union[str, bytes] if image is in Base64 format or raw? format then use instead of filename
location	Union[Tuple[int, int], Tuple[float, float]] the (x,y) location to place image's top left corner
color	(str) text color
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
angle	(float) Angle 0 to 360 to draw the text. Zero represents horizontal text
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the image

draw_line

Draws a line from one point to another point using USER'S coordinates. Can set the color and width of line

```
draw_line(point_from,
    point_to,
    color="black",
    width=1)
```

Parameter Descriptions:

Name	Meaning
point_from	Union[Tuple[int, int], Tuple[float, float]] Starting point for line
point_to	Union[Tuple[int, int], Tuple[float, float]] Ending point for line
color	(str) Color of the line
width	(int) width of line in pixels
return	Union[int, None] id returned from tktiner or None if user closed the window. id is used when you want to manipulate the line

draw_oval

Draws an oval based on coordinates in user coordinate system. Provide the location of a "bounding rectangle"

```
draw_oval(top_left,
    bottom_right,
    fill_color=None,
    line_color=None)
```

Parameter Descriptions:

Name	Meaning
top_left	Union[Tuple[int, int], Tuple[float, float]] the top left point of bounding rectangle
bottom_right	Union[Tuple[int, int], Tuple[float, float]] the bottom right point of bounding rectangle
fill_color	(str) color of the interrior
line_color	(str) color of outline of oval

Name	Meaning
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the oval

draw_point

Draws a "dot" at the point you specify using the USER'S coordinate system

```
draw_point(point,
    size=2,
    color="black")
```

Parameter Descriptions:

Name	Meaning
point	Union [Tuple[int, int], Tuple[float, float]] Center location using USER'S coordinate system
size	Union[int, float] Radius? (Or is it the diameter?) in user's coordinate values.
color	(str) color of the point to draw
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the point

draw_rectangle

Draw a rectangle given 2 points. Can control the line and fill colors

```
draw_rectangle(top_left,
    bottom_right,
    fill_color=None,
    line_color=None,
    line_width=None)
```

Parameter Descriptions:

Name	Meaning
top_left	Union[Tuple[int, int], Tuple[float, float]] the top left point of rectangle

Name	Meaning
bottom_right	Union[Tuple[int, int], Tuple[float, float]] the bottom right point of rectangle
fill_color	(str) color of the interior
line_color	(str) color of outline
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the rectangle

draw_text

Draw some text on your graph. This is how you label graph number lines for example

```
draw_text(text,
    location,
    color="black",
    font=None,
    angle=0,
    text_location="center")
```

Parameter Descriptions:

Name	Meaning
text	(str) text to display
location	Union[Tuple[int, int], Tuple[float, float]] location to place first letter
color	(str) text color
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
angle	(float) Angle 0 to 360 to draw the text. Zero represents horizontal text
text_location	(enum) "anchor" location for the text. Values start with TEXT_LOCATION_
return	Union[int, None] id returned from tkinter that you'll need if you want to manipulate the text

erase

Erase the Graph - Removes all figures previously "drawn" using the Graph methods (e.g. DrawText)

```
erase()
```

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

motion_call_back

Not a user callable method. Used to get Graph mouse motion events. Called by tkinter when mouse moved

```
motion_call_back(event)
```

Parameter Descriptions:

Name	Meaning
event	(event) event info from tkinter. Contains the x and y coordinates of a mouse

move

Moves the entire drawing area (the canvas) by some delta from the current position. Units are indicated in your coordinate system indicated number of ticks in your coordinate system

```
move(x_direction, y_direction)
```

Parameter Descriptions:

Name	Meaning
x_direction	Union[int, float] how far to move in the "X" direction in your coordinates
y_direction	Union[int, float] how far to move in the "Y" direction in your coordinates

move_figure

Moves a previously drawn figure using a "delta" from current position

```
move_figure(figure,
    x_direction,
    y_direction)
```

Parameter Descriptions:

Name	Meaning
figure	(id) Previously obtained figure-id. These are returned from all Draw methods
x_direction	Union[int, float] delta to apply to position in the X direction
y_direction	Union[int, float] delta to apply to position in the Y direction

relocate_figure

Move a previously made figure to an arbitrary (x,y) location. This differs from the Move methods because it uses absolute coordinates versus relative for Move

```
relocate_figure(figure,
    x,
    y)
```

Parameter Descriptions:

Name	Meaning
figure	(id) Previously obtained figure-id. These are returned from all Draw methods
x	Union[int, float] location on X axis (in user coords) to move the upper left corner of the figure
y	Union[int, float] location on Y axis (in user coords) to move the upper left corner of the figure

send_figure_to_back

Changes Z-order of figures on the Graph. Sends the indicated figure to the back of all other drawn figures

```
send_figure_to_back(figure)
```

Parameter Descriptions:

Name	Meaning
figure	(int) value returned by tkinter when creating the figure / drawing

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

tk_canvas

property: tk_canvas

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Graph Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(background_color=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
background_color	color of background
visible	(bool) control visibility of element

Image Element

Image Element - show an image in the window. Should be a GIF or a PNG only

```
Image(filename=None,
    data=None,
    background_color=None,
    size=(None, None),
    pad=None,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    enable_events=False,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
filename	(str) image filename if there is a button image. GIFs and PNGs only.
data	Union[bytes, str] Raw or Base64 representation of the image to put on button. Choose either filename or data
background_color	color of background
size	Tuple[int, int] (width, height) size of image in pixels

Name	Meaning
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
enable_events	(bool) Turns on the element specific events. For an Image element, the event is "image clicked"
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Image Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(filename=None,
    data=None,
    size=(None, None),
    visible=None)
```

Parameter Descriptions:

Name	Meaning
filename	(str) filename to the new image to display.
data	Union[str, tkPhotoImage] Base64 encoded string OR a tk.PhotoImage object
size	Tuple[int,int] size of a image (w,h) w=characters-wide, h=rows-high
visible	(bool) control visibility of element

UpdateAnimation

Show an Animated GIF. Call the function as often as you like. The function will determine when

to show the next frame and will automatically advance to the next frame at the right time. NOTE - does NOT perform a sleep call to delay

```
UpdateAnimation(source, time_between_frames=0)
```

Parameter Descriptions:

Name	Meaning
source	Union[str,bytes] Filename or Base64 encoded string containing Animated GIF
time_between_frames	(int) Number of milliseconds to wait between showing frames

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions

Name	Meaning
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Image Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(filename=None,
    data=None,
    size=(None, None),
    visible=None)
```

Parameter Descriptions:

Name	Meaning
filename	(str) filename to the new image to display.
data	Union[str, tkPhotoImage] Base64 encoded string OR a tk.PhotoImage object

Name	Meaning
size	Tuple[int,int] size of a image (w,h) w=characters-wide, h=rows-high
visible	(bool) control visibility of element

update_animation

Show an Animated GIF. Call the function as often as you like. The function will determine when to show the next frame and will automatically advance to the next frame at the right time. NOTE - does NOT perform a sleep call to delay

```
update_animation(source, time_between_frames=0)
```

Parameter Descriptions:

Name	Meaning
source	Union[str,bytes] Filename or Base64 encoded string containing Animated GIF
time_between_frames	(int) Number of milliseconds to wait between showing frames

InputText Element

```
Display a single text input field.  Based on the tkinter Widget `Entry`

InputText(default_text="",
    size=(None, None),
    disabled=False,
    password_char="",
    justification=None,
    background_color=None,
    text_color=None,
    font=None,
    tooltip=None,
    change_submits=False,
    enable_events=False,
    do_not_clear=True,
    key=None,
    focus=False,
    pad=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
default_text	(str) Text initially shown in the input box as a default value(Default value = '')
size	Tuple[int, int] (width, height) w=characters-wide, h=rows-high
disabled	(bool) set disable state for element (Default = False)
password_char	(char) Password character if this is a password field (Default value = '')
justification	(str) justification for data display. Valid choices - left, right, center
background_color	(str) color of background in one of the color formats
text_color	(str) color of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
tooltip	(str) text, that will appear when mouse hovers over the element
change_submits	(bool) * DEPRICATED DO NOT USE! Same as enable_events
enable_events	(bool) If True then changes to this element are immediately reported as an event. Use this instead of change_submits (Default = False)
do_not_clear	(bool) If False then the field will be set to blank after ANY event (button, any event) (Default = True)
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
focus	(bool) Determines if initial focus should go to this element.
pad	(int, int) or ((int, int), (int, int)) Tuple(s). Amount of padding to put around element. Normally (horizontal pixels, vertical pixels) but can be split apart further into ((horizontal left, horizontal right), (vertical above, vertical below))
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.

Name	Meaning
visible	(bool) set visibility state of the element (Default = True)
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Read and return the current value of the input element. Must call `Window.Read` or `Window.Finalize` prior

`Get()`

Name	Meaning
return	(str) current value of Input field or '' if error encountered

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Input Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    disabled=None,
    select=None,
    visible=None,
    text_color=None,
    background_color=None,
    move_cursor_to="end")
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to display as default text in Input field
disabled	(bool) disable or enable state of the element (sets Entry Widget to readonly or normal)
select	(bool) if True, then the text will be selected
visible	(bool) change visibility of element

Name	Meaning
text_color	(str) change color of text being typed
background_color	(str) change color of the background
move_cursor_to	Union[int, str] Moves the cursor to a particular offset. Defaults to 'end'

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

Read and return the current value of the input element. Must call `Window.Read` or `Window.Finalize` prior

`get()`

Name	Meaning
return	(str) current value of Input field or '' if error encountered

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Input Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    disabled=None,
    select=None,
    visible=None,
    text_color=None,
    background_color=None,
    move_cursor_to="end")
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to display as default text in Input field
disabled	(bool) disable or enable state of the element (sets Entry Widget to readonly or normal)
select	(bool) if True, then the text will be selected
visible	(bool) change visibility of element
text_color	(str) change color of text being typed
background_color	(str) change color of the background
move_cursor_to	Union[int, str] Moves the cursor to a particular offset. Defaults to 'end'

Listbox Element

A List Box. Provide a list of values for the user to choose one or more of. Returns a list of selected rows when a window.Read() is executed.

```
Listbox(values,
    default_values=None,
    select_mode=None,
    change_submits=False,
    enable_events=False,
    bind_return_key=False,
    size=(None, None),
    disabled=False,
    auto_size_text=None,
    font=None,
    no_scrollbar=False,
    background_color=None,
    text_color=None,
    key=None,
    pad=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] list of values to display. Can be any type including mixed types as long as they have **str** method
default_values	List[Any] which values should be initially selected
select_mode	[enum] Select modes are used to determine if only 1 item can be selected or multiple and how they can be selected. Valid choices begin with "LISTBOX_SELECT_MODE_" and include: LISTBOX_SELECT_MODE_SINGLE LISTBOX_SELECT_MODE_MULTIPLE LISTBOX_SELECT_MODE_BROWSE LISTBOX_SELECT_MODE_EXTENDED
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Listbox generates events when an item is clicked
bind_return_key	(bool) If True, then the return key will cause a the Listbox to generate an event
size	Tuple(int, int) (width, height) width = characters-wide, height = rows-high
disabled	(bool) set disable state for element
auto_size_text	(bool) True if element should be the same size as the contents
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
background_color	(str) color of background
text_color	(str) color of the text
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text, that will appear when mouse hovers over the element

Name	Meaning
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

GetIndexes

Returns the items currently selected as a list of indexes

`GetIndexes()`

Name	Meaning
return	List[int] A list of offsets into values that is currently selected

GetListValues

Returns list of Values provided by the user in the user's format

`GetListValues()`

Name	Meaning
return	List[Any]. List of values. Can be any / mixed types -> []

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

SetValue

Set listbox highlighted choices

```
SetValue(values)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] new values to choose based on previously set values

Update

Changes some of the settings for the Listbox Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(values=None,
    disabled=None,
    set_to_index=None,
    scroll_to_index=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] new list of choices to be shown to user
disabled	(bool) disable or enable state of the element
set_to_index	Union[int, list, tuple] highlights the item(s) indicated. If parm is an int one entry will be set. If is a list, then each entry in list is highlighted
scroll_to_index	(int) scroll the listbox so that this index is the first shown
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_indexes

Returns the items currently selected as a list of indexes

`get_indexes()`

Name	Meaning
return	List[int] A list of offsets into values that is currently selected

get_list_values

Returns list of Values provided by the user in the user's format

`get_list_values()`

Name	Meaning
return	List[Any]. List of values. Can be any / mixed types -> []

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

set_value

Set listbox highlighted choices

```
set_value(values)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] new values to choose based on previously set values

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Listbox Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(values=None,
    disabled=None,
    set_to_index=None,
    scroll_to_index=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] new list of choices to be shown to user

Name	Meaning
disabled	(bool) disable or enable state of the element
set_to_index	Union[int, list, tuple] highlights the item(s) indicated. If parm is an int one entry will be set. If is a list, then each entry in list is highlighted
scroll_to_index	(int) scroll the listbox so that this index is the first shown
visible	(bool) control visibility of element

Menu Element is the Element that provides a Menu Bar that goes across the top of the window, just below titlebar.
Here is an example layout. The "&" are shortcut keys ALT+key.
Is a List of - "Item String" + List
Where Item String is what will be displayed on the Menubar itself.
The List that follows the item represents the items that are shown then Menu item is clicked
Notice how an "entry" in a mennu can be a list which means it branches out and shows another menu, etc. (resursive)
menu_def = [['&File', ['!&Open', '&Save::savekey', '---', '&Properties', 'E&xit']],
['!&Edit', ['!&Paste', ['Special', 'Normal',], 'Undo'],],
['&Debugger', ['Popout', 'Launch Debugger']],
['&Toolbar', ['Command &1', 'Command &2', 'Command &3', 'Command &4']],
['&Help', '&About...'],]
Finally, "keys" can be added to entries so make them unique. The "Save" entry has a key associated with it. You
can see it has a "::" which signifies the beginning of a key. The user will not see the key portion when the menu is shown. The key portion is returned as part of the event.

```
Menu(menu_definition,
    background_color=None,
    size=(None, None),
    tearoff=False,
    pad=None,
    key=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
menu_definition	List[List[Tuple[str, List[str]]]
background_color	(str) color of the background
size	Tuple[int, int] Not used in the tkinter port

Name	Meaning
tearoff	(bool) if True, then can tear the menu off from the window ans use as a floating window. Very cool effect
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Update a menubar - can change the menu definition and visibility. The entire menu has to be specified

```
Update(menu_definition=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
menu_definition	List[List[Tuple[str, List[str]]]
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

set_focus(force=False)

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

set_size(size=(None, None))

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

set_tooltip(tooltip_text)

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

unhide_row()

update

Update a menubar - can change the menu definition and visibility. The entire menu has to be specified

```
update(menu_definition=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
menu_definition	List[List[Tuple[str, List[str]]]
visible	(bool) control visibility of element

Multiline Element

Multiline Element - Display and/or read multiple lines of text. This is both an input and output element. Other PySimpleGUI ports have a separate MultilineInput and MultilineOutput elements. May want to split this one up in the future too.

```
Multiline(default_text="",
    enter_submits=False,
    disabled=False,
    autoscroll=False,
    border_width=None,
    size=(None, None),
    auto_size_text=None,
    background_color=None,
    text_color=None,
    change_submits=False,
    enable_events=False,
    do_not_clear=True,
    key=None,
    focus=False,
    font=None,
    pad=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
default_text	(str) Initial text to show
enter_submits	(bool) if True, the Window.Read call will return is enter key is pressed in this element
disabled	(bool) set disable state
autoscroll	(bool) If True the contents of the element will automatically scroll as more data added to the end
border_width	(int) width of border around element in pixels
size	Tuple[int, int] (width, height) width = characters-wide, height = rows-high
auto_size_text	(bool) if True will size the element to match the length of the text
background_color	(str) color of background
text_color	(str) color of the text
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Spin events happen when an item changes
do_not_clear	if False the element will be cleared any time the Window.Read call returns
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
focus	(bool) if True initial focus will go to this element
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.

Name	Meaning
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Return current contents of the Multiline Element

`Get()`

Name	Meaning
return	(str) current contents of the Multiline Element (used as an input type of Multiline

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Multiline Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    disabled=None,
    append=False,
    font=None,
    text_color=None,
    background_color=None,
    visible=None,
    autoscroll=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to display
disabled	(bool) disable or enable state of the element
append	(bool) if True then new value will be added onto the end of the current value. if False then contents will be replaced.
font	Union[str, Tuple[str, int]] specifies the font family, size, etc

Name	Meaning
text_color	(str) color of the text
background_color	(str) color of background
visible	(bool) set visibility state of the element
autoscroll	(bool) if True then contents of element are scrolled down when new text is added to the end

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

Return current contents of the Multiline Element

`get()`

Name	Meaning
return	(str) current contents of the Multiline Element (used as an input type of Multiline

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Multiline Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    disabled=None,
    append=False,
    font=None,
    text_color=None,
    background_color=None,
    visible=None,
    autoscroll=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to display
disabled	(bool) disable or enable state of the element
append	(bool) if True then new value will be added onto the end of the current value. if False then contents will be replaced.
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
text_color	(str) color of the text
background_color	(str) color of background
visible	(bool) set visibility state of the element
autoscroll	(bool) if True then contents of element are scrolled down when new text is added to the end

Option Menu is an Element available ONLY on the tkinter port of PySimpleGUI. It's is a widget that is unique to tkinter. However, it looks much like a ComboBox. Instead of an arrow to click to pull down the list of choices, another little graphic is shown on the widget to indicate where you click. After clicking to activate, it looks like a Combo Box that you scroll to select a choice.

```
OptionMenu(values,
    default_value=None,
    size=(None, None),
    disabled=False,
    auto_size_text=None,
    background_color=None,
    text_color=None,
    key=None,
    pad=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] Values to be displayed
default_value	(Any) the value to choose by default

Name	Meaning
size	Tuple[int, int] (width, height) size in characters (wide) and rows (high)
disabled	(bool) control enabled / disabled
auto_size_text	(bool) True if size of Element should match the contents of the items
background_color	(str) color of background
text_color	(str) color of the text
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text that will appear when mouse hovers over this element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the OptionMenu Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    values=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) the value to choose by default
values	List[Any] Values to be displayed
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the OptionMenu Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    values=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) the value to choose by default
values	List[Any] Values to be displayed
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

Output Element

Output Element - a multi-lined text area where stdout and stderr are re-routed to.

```
Output(size=(None, None),
    background_color=None,
    text_color=None,
    pad=None,
    font=None,
    tooltip=None,
    key=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
background_color	(str) color of background
text_color	(str) color of the text
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
tooltip	(str) text, that will appear when mouse hovers over the element
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Returns the current contents of the output. Similar to Get method other Elements

`Get()`

Name	Meaning
return	(str) the current value of the output

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

TKOut

property: TKOut

Update

Changes some of the settings for the Output Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) string that will replace current contents of the output area
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

set_size(size=(None, None))

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

set_tooltip(tooltip_text)

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

tk_out

property: tk_out

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

unhide_row()

update

Changes some of the settings for the Output Element. Must call `Window.Read` or `Window.Finalize` prior

update(value=None, visible=None)

Parameter Descriptions:

Name	Meaning
value	(str) string that will replace current contents of the output area
visible	(bool) control visibility of element

Pane Element

A sliding Pane that is unique to tkinter. Uses Columns to create individual panes

```
Pane(pane_list,
    background_color=None,
    size=(None, None),
    pad=None,
    orientation="vertical",
    show_handle=True,
    relief="raised",
    handle_size=None,
    border_width=None,
    key=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
pane_list	List[Column] Must be a list of Column Elements. Each Column supplied becomes one pane that's shown
background_color	(str) color of background
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high How much room to reserve for the Pane
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
orientation	(str) 'horizontal' or 'vertical' or ('h' or 'v'). Direction the Pane should slide
show_handle	(bool) if True, the handle is drawn that makes it easier to grab and slide

Name	Meaning
relief	(enum) relief style. Values are same as other elements that use relief values. RELIEF_RAISED RELIEF_SUNKEN RELIEF_FLAT RELIEF_RIDGE RELIEF_GROOVE RELIEF_SOLID
handle_size	(int) Size of the handle in pixels
border_width	(int) width of border around element in pixels
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Pane Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(visible=None)
```

Parameter Descriptions:

Name	Meaning
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

set_focus(force=False)

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

set_size(size=(None, None))

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

set_tooltip(tooltip_text)

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

unhide_row()

update

Changes some of the settings for the Pane Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(visible=None)
```

Parameter Descriptions:

Name	Meaning
visible	(bool) control visibility of element

ProgressBar Element

Progress Bar Element - Displays a colored bar that is shaded as progress of some operation is made

```
ProgressBar(max_value,
    orientation=None,
    size=(None, None),
    auto_size_text=None,
    bar_color=(None, None),
    style=None,
    border_width=None,
    relief=None,
    key=None,
    pad=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
max_value	(int) max value of progressbar
orientation	(str) 'horizontal' or 'vertical'
size	Tuple[int, int] Size of the bar. If horizontal (chars wide, pixels high), vert (pixels wide, rows high)
auto_size_text	(bool) Not sure why this is here
bar_color	Tuple[str, str] The 2 colors that make up a progress bar. One is the background, the other is the bar

Name	Meaning
style	(str) Progress bar style defined as one of these 'default', 'winnative', 'clam', 'alt', 'classic', 'vista', 'xpnative'
border_width	(int) The amount of pixels that go around the outside of the bar
relief	(str) relief style. Values are same as progress meter relief values. Can be a constant or a string: `RELIEF_RAISED RELIEF_SUNKEN RELIEF_FLAT RELIEF_RIDGE RELIEF_GROOVE RELIEF_SOLID` (Default value = DEFAULT_PROGRESS_BAR_RELIEF)
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the ProgressBar Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(visible=None)
```

Parameter Descriptions:

Name	Meaning
visible	(bool) control visibility of element

UpdateBar

Change what the bar shows by changing the current count and optionally the max count

```
UpdateBar(current_count, max=None)
```

Parameter Descriptions:

Name	Meaning
current_count	(int) sets the current value

Name	Meaning
max	(int) changes the max value

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

set_tooltip(tooltip_text)

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

unhide_row()

update

Changes some of the settings for the ProgressBar Element. Must call `Window.Read` or `Window.Finalize` prior

update(visible=None)

Parameter Descriptions:

Name	Meaning
visible	(bool) control visibility of element

update_bar

Change what the bar shows by changing the current count and optionally the max count

update_bar(current_count, max=None)

Parameter Descriptions:

Name	Meaning
current_count	(int) sets the current value
max	(int) changes the max value

Radio Element

Radio Button Element - Used in a group of other Radio Elements to provide user with ability to select only 1 choice in a list of choices.

```
Radio(text,
    group_id,
    default=False,
    disabled=False,
    size=(None, None),
    auto_size_text=None,
    background_color=None,
    text_color=None,
    font=None,
    key=None,
    pad=None,
    tooltip=None,
    change_submits=False,
    enable_events=False,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
text	(str) Text to display next to button
group_id	(Any) Groups together multiple Radio Buttons. Any type works
default	(bool). Set to True for the one element of the group you want initially selected
disabled	(bool) set disable state
size	Tuple[int, int] (width, height) width = characters-wide, height = rows-high
auto_size_text	(bool) if True will size the element to match the length of the text
background_color	(str) color of background
text_color	(str) color of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
key	(Any) Used with window.FindElement and with return values to uniquely identify this element

Name	Meaning
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text, that will appear when mouse hovers over the element
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Radio Button events happen when an item is selected
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

A snapshot of the value of Radio Button -> (bool)

`Get()`

Name	Meaning
return	(bool) True if this radio button is selected

ResetGroup

Sets all Radio Buttons in the group to not selected

```
ResetGroup()
```

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Radio Button Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(bool) if True change to selected and set others in group to unselected
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

A snapshot of the value of Radio Button -> (bool)

`get()`

Name	Meaning
return	(bool) True if this radio button is selected

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

reset_group

Sets all Radio Buttons in the group to not selected

```
reset_group()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Radio Button Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(bool) if True change to selected and set others in group to unselected
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

Slider Element

A slider, horizontal or vertical

```
Slider(range=(None, None),
    default_value=None,
    resolution=None,
    tick_interval=None,
    orientation=None,
    disable_number_display=False,
    border_width=None,
    relief=None,
    change_submits=False,
    enable_events=False,
    disabled=False,
    size=(None, None),
    font=None,
    background_color=None,
    text_color=None,
    key=None,
    pad=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
range	Union[Tuple[int, int], Tuple[float, float]] slider's range (min value, max value)
default_value	Union[int, float] starting value for the slider
resolution	Union[int, float] the smallest amount the slider can be moved

Name	Meaning
tick_interval	Union[int, float] how often a visible tick should be shown next to slider
orientation	(str) 'horizontal' or 'vertical' ('h' or 'v' also work)
disable_number_display	(bool) if True no number will be displayed by the Slider Element
border_width	(int) width of border around element in pixels
relief	(enum) relief style. RELIEF_RAISED RELIEF_SUNKEN RELIEF_FLAT RELIEF_RIDGE RELIEF_GROOVE RELIEF_SOLID
change_submits	(bool) * DEPRICATED DO NOT USE! Same as enable_events
enable_events	(bool) If True then moving the slider will generate an Event
disabled	(bool) set disable state for element
size	Tuple[int, int] (width in characters, height in rows)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
background_color	(str) color of slider's background
text_color	(str) color of the slider's text
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text, that will appear when mouse hovers over the element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Slider Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    range=(None, None),
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	Union[int, float] sets current slider value
range	Union[Tuple[int, int], Tuple[float, float] Sets a new range for slider
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

```
get_size()
```

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

set_size(size=(None, None))

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

set_tooltip(tooltip_text)

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

unhide_row()

update

Changes some of the settings for the Slider Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
   range=(None, None),
   disabled=None,
   visible=None)
```

Parameter Descriptions:

Name	Meaning
value	Union[int, float] sets current slider value

Name	Meaning
range	Union[Tuple[int, int], Tuple[float, float] Sets a new range for slider
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

Spin Element

A spinner with up/down buttons and a single line of text. Choose 1 values from list

```
Spin(values,
    initial_value=None,
    disabled=False,
    change_submits=False,
    enable_events=False,
    size=(None, None),
    auto_size_text=None,
    font=None,
    background_color=None,
    text_color=None,
    key=None,
    pad=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
values	List[Any] List of valid values
initial_value	(Any) Initial item to show in window. Choose from list of values supplied
disabled	(bool) set disable state
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Spin events happen when an item changes

Name	Meaning
size	Tuple[int, int] (width, height) width = characters-wide, height = rows-high
auto_size_text	(bool) if True will size the element to match the length of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
background_color	(str) color of background
text_color	(str) color of the text
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
tooltip	(str) text, that will appear when mouse hovers over the element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Return the current chosen value showing in spinbox. This value will be the same as what was provided as list of choices. If list items are ints, then the item returned will be an int (not a string)

`Get()`

Name	Meaning
return	(Any) The currently visible entry

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Spin Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    values=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) set the current value from list of choices
values	List[Any] set available choices
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

Return the current chosen value showing in spinbox. This value will be the same as what was provided as list of choices. If list items are ints, then the item returned will be an int (not a string)

`get()`

Name	Meaning
return	(Any) The currently visible entry

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Spin Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    values=None,
    disabled=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(Any) set the current value from list of choices
values	List[Any] set available choices
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

StatusBar Element

A StatusBar Element creates the sunken text-filled strip at the bottom. Many Windows programs have this line

```
StatusBar(text,
    size=(None, None),
    auto_size_text=None,
    click_submits=None,
    enable_events=False,
    relief="sunken",
    font=None,
    text_color=None,
    background_color=None,
    justification=None,
    pad=None,
    key=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
text	(str) Text that is to be displayed in the widget

Name	Meaning
size	Tuple[(int), (int)] (w,h) w=characters-wide, h=rows-high
auto_size_text	(bool) True if size should fit the text length
click_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. StatusBar events occur when the bar is clicked
relief	(enum) relief style. Values are same as progress meter relief values. Can be a constant or a string: `RELIEF_RAISED RELIEF_SUNKEN RELIEF_FLAT RELIEF_RIDGE RELIEF_GROOVE RELIEF_SOLID`
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
text_color	(str) color of the text
background_color	(str) color of background
justification	(str) how string should be aligned within space provided by size. Valid choices = `left`, `right`, `center`
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
tooltip	(str) text, that will appear when mouse hovers over the element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Status Bar Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    background_color=None,
    text_color=None,
    font=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to show
background_color	(str) color of background
text_color	(str) color of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) set visibility state of the element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions

Name	Meaning
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Status Bar Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
  background_color=None,
  text_color=None,
  font=None,
  visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to show
background_color	(str) color of background

Name	Meaning
text_color	(str) color of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) set visibility state of the element

Tab Element

Tab Element is another "Container" element that holds a layout and displays a tab with text. Used with TabGroup only
Tabs are never placed directly into a layout. They are always "Contained" in a TabGroup layout

```
Tab(title,
    layout,
    title_color=None,
    background_color=None,
    font=None,
    pad=None,
    disabled=False,
    border_width=None,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    element_justification="left",
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
title	(str) text to show on the tab
layout	List[List[Element]] The element layout that will be shown in the tab
title_color	(str) color of the tab text (note not currently working on tkinter)
background_color	(str) color of background of the entire layout
font	Union[str, Tuple[str, int]] specifies the font family, size, etc

Name	Meaning
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
disabled	(bool) If True button will be created disabled
border_width	(int) width of border around element in pixels
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
element_justification	(str) All elements inside the Tab will have this justification 'left', 'right', 'center' are valid values
metadata	(Any) User metadata that can be set to ANYTHING

AddRow

Not recommended use call. Used to add rows of Elements to the Frame Element.

```
AddRow(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback

function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Layout

Not user callable. Use layout parameter instead. Creates the layout using the supplied rows of Elements

```
Layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The list of rows
return	(Tab) used for chaining

Select

Create a tkinter event that mimics user clicking on a tab. Must have called window.Finalize / Read first!

```
Select()
```

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Tab Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(disabled=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

add_row

Not recommended use call. Used to add rows of Elements to the Frame Element.

```
add_row(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

layout

Not user callable. Use layout parameter instead. Creates the layout using the supplied rows of Elements

```
layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The list of rows
return	(Tab) used for chaining

select

Create a tkinter event that mimics user clicking on a tab. Must have called window.Finalize / Read first!

```
select()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Tab Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(disabled=None, visible=None)
```

Parameter Descriptions:

Name	Meaning
disabled	(bool) disable or enable state of the element
visible	(bool) control visibility of element

TabGroup Element

TabGroup Element groups together your tabs into the group of tabs you see displayed in your window

```
TabGroup(layout,
    tab_location=None,
    title_color=None,
    selected_title_color=None,
    background_color=None,
    font=None,
    change_submits=False,
    enable_events=False,
    pad=None,
    border_width=None,
    theme=None,
    key=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
layout	List[List[Tab]] Layout of Tabs. Different than normal layouts. ALL Tabs should be on first row
tab_location	(str) location that tabs will be displayed. Choices are left, right, top, bottom, lefttop, leftbottom, righttop, rightbottom, bottomleft, bottomright, topleft, topright
title_color	(str) color of text on tabs
selected_title_color	(str) color of tab when it is selected

Name	Meaning
background_color	(str) color of background of tabs
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
change_submits	(bool) * DEPRICATED DO NOT USE! Same as enable_events
enable_events	(bool) If True then switching tabs will generate an Event
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
border_width	(int) width of border around element in pixels
theme	(enum) tabs can be 'themed'. These are the choices (some may not work on your OS): THEME_DEFAULT THEME_WINNATIVE THEME_CLAM THEME_ALT THEME_CLASSIC THEME_VISTA THEME_XPNATIVE
key	(any) Value that uniquely identifies this element from all other elements. Used when Finding an element or in return values. Must be unique to the window
tooltip	(str) text, that will appear when mouse hovers over the element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

AddRow

Not recommended user call. Used to add rows of Elements to the Frame Element.

```
AddRow(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

FindKeyFromTabName

Searches through the layout to find the key that matches the text on the tab. Implies names should be unique

```
FindKeyFromTabName(tab_name)
```

Parameter Descriptions:

Name	Meaning
tab_name	
return	Union[key, None] Returns the key or None if no key found

Get

Returns the current value for the Tab Group, which will be the currently selected tab's KEY or the text on the tab if no key is defined. Returns None if an error occurs. Note that this is exactly the same data that would be returned from a call to Window.Read. Are you sure you are using this method correctly?

`Get()`

Name	Meaning
return	Union[Any, None] The key of the currently selected tab or the tab's text if it has no key

Layout

Can use like the Window.Layout method, but it's better to use the layout parameter when creating

```
Layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The rows of Elements
return	(Frame) Used for chaining

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

add_row

Not recommended user call. Used to add rows of Elements to the Frame Element.

```
add_row(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Element] The list of elements for this row

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

find_key_from_tab_name

Searches through the layout to find the key that matches the text on the tab. Implies names should be unique

```
find_key_from_tab_name(tab_name)
```

Parameter Descriptions:

Name	Meaning
tab_name	
return	Union[key, None] Returns the key or None if no key found

get

Returns the current value for the Tab Group, which will be the currently selected tab's KEY or the text on the tab if no key is defined. Returns None if an error occurs. Note that this is exactly the same data that would be returned from a call to Window.Read. Are you sure you are using this method correctly?

`get()`

Name	Meaning
return	Union[Any, None] The key of the currently selected tab or the tab's text if it has no key

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

layout

Can use like the Window.Layout method, but it's better to use the layout parameter when creating

```
layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Element]] The rows of Elements
return	(Frame) Used for chaining

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

Table Element

```
Table(values,
    headings=None,
    visible_column_map=None,
    col_widths=None,
    def_col_width=10,
    auto_size_columns=True,
    max_col_width=20,
    select_mode=None,
    display_row_numbers=False,
    num_rows=None,
    row_height=None,
    font=None,
    justification="right",
    text_color=None,
    background_color=None,
    alternating_row_color=None,
    row_colors=None,
    vertical_scroll_only=True,
    hide_vertical_scroll=False,
    size=(None, None),
    change_submits=False,
    enable_events=False,
    bind_return_key=False,
    pad=None,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
values	List[List[Union[str, int, float]]]
headings	List[str] The headings to show on the top line
visible_column_map	List[bool] One entry for each column. False indicates the column is not shown
col_widths	List[int] Number of characters that each column will occupy
def_col_width	(int) Default column width in characters
auto_size_columns	(bool) if True columns will be sized automatically
max_col_width	(int) Maximum width for all columns in characters

Name	Meaning
select_mode	(enum) Select Mode. Valid values start with "TABLE_SELECT_MODE_". Valid values are: TABLE_SELECT_MODE_NONE TABLE_SELECT_MODE_BROWSE TABLE_SELECT_MODE_EXTENDED
display_row_numbers	(bool) if True, the first column of the table will be the row #
num_rows	(int) The number of rows of the table to display at a time
row_height	(int) height of a single row in pixels
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
justification	(str) 'left', 'right', 'center' are valid choices
text_color	(str) color of the text
background_color	(str) color of background
alternating_row_color	(str) if set then every other row will have this color in the background.
row_colors	
vertical_scroll_only	(bool) if True only the vertical scrollbar will be visible
hide_vertical_scroll	(bool) if True vertical scrollbar will be hidden
size	Tuple[int, int] DO NOT USE! Use num_rows instead
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Table events happen when row is clicked
bind_return_key	(bool) if True, pressing return key will cause event coming from Table, ALSO a left button double click will generate an event if this parameter is True
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

Get

Dummy function for tkinter port. In the Qt port you can read back the values in the table in case they were edited. Don't know yet how to enable editing of a Tree in tkinter so just returning the values provided by user when Table was created or Updated.

`Get()`

Name	Meaning

Name	Meaning
return	List[List[Any]] the current table values (for now what was originally provided up updated)

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Table Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(values=None,
    num_rows=None,
    visible=None,
    select_rows=None,
    alternating_row_color=None,
    row_colors=None)
```

Parameter Descriptions:

Name	Meaning
values	List[List[Union[str, int, float]]] A new 2-dimensional table to show
num_rows	(int) How many rows to display at a time
visible	(bool) if True then will be visible
select_rows	List[int] List of rows to select as if user did
alternating_row_color	(str) the color to make every other row
row_colors	List[Union[Tuple[int, str], Tuple[Int, str, str]] list of tuples of (row, background color) OR (row, foreground color, background color). Changes the colors of listed rows to the color(s) provided (note the optional foreground color)

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get

Dummy function for tkinter port. In the Qt port you can read back the values in the table in case they were edited. Don't know yet how to enable editing of a Tree in tkinter so just returning the values provided by user when Table was created or Updated.

`get()`

Name	Meaning
return	List[List[Any]] the current table values (for now what was originally provided up updated)

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

treeview_double_click

Not user callable. Callback function that is called when something is selected from Table. Stores the selected rows in Element as they are being selected. If events enabled, then returns from Read

```
treeview_double_click(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) event information from tkinter

treeview_selected

Not user callable. Callback function that is called when something is selected from Table. Stores the selected rows in Element as they are being selected. If events enabled, then returns from Read

```
treeview_selected(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) event information from tkinter

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Table Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(values=None,
    num_rows=None,
    visible=None,
    select_rows=None,
    alternating_row_color=None,
    row_colors=None)
```

Parameter Descriptions:

Name	Meaning
values	List[List[Union[str, int, float]]] A new 2-dimensional table to show

Name	Meaning
num_rows	(int) How many rows to display at a time
visible	(bool) if True then will be visible
select_rows	List[int] List of rows to select as if user did
alternating_row_color	(str) the color to make every other row
row_colors	List[Union[Tuple[int, str], Tuple[Int, str, str]] list of tuples of (row, background color) OR (row, foreground color, background color). Changes the colors of listed rows to the color(s) provided (note the optional foreground color)

Text Element

Text - Display some text in the window. Usually this means a single line of text. However, the text can also be multiple lines. If multi-lined there are no scroll bars.

```
Text(text="",
    size=(None, None),
    auto_size_text=None,
    click_submits=False,
    enable_events=False,
    relief=None,
    font=None,
    text_color=None,
    background_color=None,
    border_width=None,
    justification=None,
    pad=None,
    key=None,
    right_click_menu=None,
    tooltip=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
text	(str) The text to display. Can include /n to achieve multiple lines
size	Tuple[int, int] (width, height) width = characters-wide, height = rows-high

Name	Meaning
auto_size_text	(bool) if True size of the Text Element will be sized to fit the string provided in 'text' parm
click_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Text events happen when the text is clicked
relief	(str/enum) relief style around the text. Values are same as progress meter relief values. Should be a constant that is defined at starting with "RELIEF_" - `RELIEF_RAISED, RELIEF_SUNKEN, RELIEF_FLAT, RELIEF_RIDGE, RELIEF_GROOVE, RELIEF_SOLID`
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
text_color	(str) color of the text
background_color	(str) color of background
border_width	(int) number of pixels for the border (if using a relief)
justification	(str) how string should be aligned within space provided by size. Valid choices = `left`, `right`, `center`
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
tooltip	(str) text, that will appear when mouse hovers over the element
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Text Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(value=None,
    background_color=None,
    text_color=None,
    font=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to show
background_color	(str) color of background
text_color	(str) color of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) set visibility state of the element

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

update

Changes some of the settings for the Text Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(value=None,
    background_color=None,
    text_color=None,
    font=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
value	(str) new text to show
background_color	(str) color of background
text_color	(str) color of the text
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
visible	(bool) set visibility state of the element

Tree Element

Tree Element - Presents data in a tree-like manner, much like a file/folder browser. Uses the TreeData class to hold the user

```
Tree(data=None,
    headings=None,
    visible_column_map=None,
    col_widths=None,
    col0_width=10,
    def_col_width=10,
    auto_size_columns=True,
    max_col_width=20,
    select_mode=None,
    show_expanded=False,
    change_submits=False,
    enable_events=False,
    font=None,
    justification="right",
    text_color=None,
    background_color=None,
    num_rows=None,
    row_height=None,
    pad=None,
    key=None,
    tooltip=None,
    right_click_menu=None,
    visible=True,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
data	(TreeData) The data represented using a PySimpleGUI provided TreeData class
headings	List[str] List of individual headings for each column
visible_column_map	List[bool] Determines if a column should be visible. If left empty, all columns will be shown
col_widths	List[int] List of column widths so that individual column widths can be controlled
col0_width	(int) Size of Column 0 which is where the row numbers will be optionally shown
def_col_width	(int) default column width
auto_size_columns	(bool) if True, the size of a column is determined using the contents of the column
max_col_width	(int) the maximum size a column can be
select_mode	(enum) Use same values as found on Table Element. Valid values include: TABLE_SELECT_MODE_NONE TABLE_SELECT_MODE_BROWSE TABLE_SELECT_MODE_EXTENDED
show_expanded	(bool) if True then the tree will be initially shown with all nodes completely expanded
change_submits	(bool) DO NOT USE. Only listed for backwards compat - Use enable_events instead
enable_events	(bool) Turns on the element specific events. Tree events happen when row is clicked
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
justification	(str) 'left', 'right', 'center' are valid choices
text_color	(str) color of the text
background_color	(str) color of background

Name	Meaning
num_rows	(int) The number of rows of the table to display at a time
row_height	(int) height of a single row in pixels
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))
key	(Any) Used with window.FindElement and with return values to uniquely identify this element to uniquely identify this element
tooltip	(str) text, that will appear when mouse hovers over the element
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
visible	(bool) set visibility state of the element
metadata	(Any) User metadata that can be set to ANYTHING

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

Update

Changes some of the settings for the Tree Element. Must call `Window.Read` or `Window.Finalize` prior

```
Update(values=None,
    key=None,
    value=None,
    text=None,
    icon=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
values	(TreeData) Representation of the tree
key	(Any) identifies a particular item in tree to update
value	(Any) sets the node identified by key to a particular value
text	(str) sets the node identified by ket to this string
icon	Union[bytes, str] can be either a base64 icon or a filename for the icon

Name	Meaning
visible	(bool) control visibility of element

add_treeview_data

Not a user function. Recursive method that inserts tree data into the tkinter treeview widget.

```
add_treeview_data(node)
```

Parameter Descriptions:

Name	Meaning
node	(TreeData) The node to insert. Will insert all nodes from starting point downward, recursively

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

```
expand(expand_x=False, expand_y=False)
```

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

set_size(size=(None, None))

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

set_tooltip(tooltip_text)

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

treeview_selected

Not a user function. Callback function that happens when an item is selected from the tree. In this method, it saves away the reported selections so they can be properly returned.

treeview_selected(event)

Parameter Descriptions:

Name	Meaning
event	(Any) An event parameter passed in by tkinter. Not used

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

unhide_row()

update

Changes some of the settings for the Tree Element. Must call `Window.Read` or `Window.Finalize` prior

```
update(values=None,
    key=None,
    value=None,
    text=None,
    icon=None,
    visible=None)
```

Parameter Descriptions:

Name	Meaning
values	(TreeData) Representation of the tree
key	(Any) identifies a particular item in tree to update
value	(Any) sets the node identified by key to a particular value
text	(str) sets the node identified by ket to this string
icon	Union[bytes, str] can be either a base64 icon or a filename for the icon
visible	(bool) control visibility of element

TreeData Element

```
Class that user fills in to represent their tree data. It
with possibly one or more children "Nodes".  Each Node contains a key, text to display, list of values to display
and an icon.  The entire tree is built using a single method, Insert.  Nothing else is required to make the tree.
```

Instantiate the object, initializes the Tree Data, creates a root node for you

```
TreeData()
```

Insert

Inserts a node into the tree. This is how user builds their tree, by Inserting Nodes This is the ONLY user callable method in the TreeData class

```
Insert(parent,
    key,
    text,
    values,
    icon=None)
```

Parameter Descriptions:

Name	Meaning
parent	(Node) the parent Node
key	(Any) Used to uniquely identify this node
text	(str) The text that is displayed at this node's location
values	List[Any] The list of values that are displayed at this node
icon	Union[str, bytes]

Node

Contains information about the individual node in the tree

```
Node(parent,
    key,
    text,
    values,
    icon=None)
```

insert

Inserts a node into the tree. This is how user builds their tree, by Inserting Nodes This is the ONLY user callable method in the TreeData class

```
insert(parent,
    key,
    text,
    values,
    icon=None)
```

Parameter Descriptions:

Name	Meaning
parent	(Node) the parent Node

Name	Meaning
key	(Any) Used to uniquely identify this node
text	(str) The text that is displayed at this node's location
values	List[Any] The list of values that are displayed at this node
icon	Union[str, bytes]

VerticalSeparator Element

Vertical Separator Element draws a vertical line at the given location. It will span 1 "row". Usually paired with Column Element if extra height is needed

```
VerticalSeparator(pad=None)
```

Parameter Descriptions:

Name	Meaning
pad	(int, int) or ((int, int),(int,int)) Amount of padding to put around element (left/right, top/bottom) or ((left, right), (top, bottom))

ButtonReboundCallback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
ButtonReboundCallback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

SetFocus

Sets the current focus to be on this element

```
SetFocus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

SetTooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
SetTooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

button_rebound_callback

Used in combination with tkinter's widget.bind function. If you wish to have a double-click for a button to call back the button's normal callback routine, then you should target your call to tkinter's bind method to point to this function which will in turn call the button callback function that is normally called.

```
button_rebound_callback(event)
```

Parameter Descriptions:

Name	Meaning
event	(unknown) Not used in this function.

expand

Causes the Element to expand to fill available space in the X and Y directions. Can specify which or both directions

expand(expand_x=False, expand_y=False)

Parameter Descriptions:

Name	Meaning
expand_x	(Bool) If True Element will expand in the Horizontal directions
expand_y	(Bool) If True Element will expand in the Vertical directions

get_size

Return the size of an element in Pixels. Care must be taken as some elements use characters to specify their size but will return pixels when calling this get_size method.

`get_size()`

Name	Meaning
return	Tuple[int, int] - Width, Height of the element

hide_row

Hide the entire row an Element is located on. Use this if you must have all space removed when you are hiding an element, including the row container

```
hide_row()
```

set_focus

Sets the current focus to be on this element

```
set_focus(force=False)
```

Parameter Descriptions:

Name	Meaning
force	(bool) if True will call focus_force otherwise calls focus_set

set_size

Changes the size of an element to a specific size. It's possible to specify None for one of sizes so that only 1 of the element's dimensions are changed.

```
set_size(size=(None, None))
```

Parameter Descriptions:

Name	Meaning
size	Tuple[int, int] The size in characters, rows typically. In some cases they are pixels

set_tooltip

Called by application to change the tooltip text for an Element. Normally invoked using the Element Object such as: window.Element('key').SetToolTip('New tip').

```
set_tooltip(tooltip_text)
```

Parameter Descriptions:

Name	Meaning
tooltip_text	(str) the text to show in tooltip.

unhide_row

Unhides (makes visible again) the row container that the Element is located on. Note that it will re-appear at the bottom of the window / container, most likely.

```
unhide_row()
```

Window

Represents a single Window

```
Window(title,
    layout=None,
    default_element_size=(45, 1),
    default_button_element_size=(None, None),
    auto_size_text=None,
    auto_size_buttons=None,
    location=(None, None),
    size=(None, None),
    element_padding=None,
    margins=(None, None),
    button_color=None,
    font=None,
    progress_bar_color=(None, None),
    background_color=None,
    border_depth=None,
    auto_close=False,
    auto_close_duration=3,
    icon=None,
    force_toplevel=False,
    alpha_channel=1,
    return_keyboard_events=False,
    use_default_focus=True,
    text_justification=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    resizable=False,
    disable_close=False,
    disable_minimize=False,
    right_click_menu=None,
    transparent_color=None,
    debugger_enabled=True,
    finalize=False,
    element_justification="left",
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
title	(str) The title that will be displayed in the Titlebar and on the Taskbar
layout	List[List[Elements]] The layout for the window. Can also be specified in the Layout method
default_element_size	Tuple[int, int] (width, height) size in characters (wide) and rows (high) for all elements in this window

Name	Meaning
default_button_element_size	Tuple[int, int] (width, height) size in characters (wide) and rows (high) for all Button elements in this window
auto_size_text	(bool) True if Elements in Window should be sized to exactly fir the length of text
auto_size_buttons	(bool) True if Buttons in this Window should be sized to exactly fit the text on this.
location	Tuple[int, int] (x,y) location, in pixels, to locate the upper left corner of the window on the screen. Default is to center on screen.
size	Tuple[int, int] (width, height) size in pixels for this window. Normally the window is autosized to fit contents, not set to an absolute size by the user
element_padding	Tuple[int, int] or ((int, int),(int,int)) Default amount of padding to put around elements in window (left/right, top/bottom) or ((left, right), (top, bottom))
margins	Tuple[int, int] (left/right, top/bottom) Amount of pixels to leave inside the window's frame around the edges before your elements are shown.
button_color	Tuple[str, str] (text color, button color) Default button colors for all buttons in the window
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
progress_bar_color	Tuple[str, str] (bar color, background color) Sets the default colors for all progress bars in the window
background_color	(str) color of background
border_depth	(int) Default border depth (width) for all elements in the window
auto_close	(bool) If True, the window will automatically close itself
auto_close_duration	(int) Number of seconds to wait before closing the window

Name	Meaning
icon	Union[str, str] Can be either a filename or Base64 value.
force_toplevel	(bool) If True will cause this window to skip the normal use of a hidden master window
alpha_channel	(float) Sets the opacity of the window. 0 = invisible 1 = completely visible. Values bewteen 0 & 1 will produce semi-transparent windows in SOME environments (The Raspberry Pi always has this value at 1 and cannot change.
return_keyboard_events	(bool) if True key presses on the keyboard will be returned as Events from Read calls
use_default_focus	(bool) If True will use the default focus algorithm to set the focus to the "Correct" element
text_justification	(str) Union ['left', 'right', 'center'] Default text justification for all Text Elements in window
no_titlebar	(bool) If true, no titlebar nor frame will be shown on window. This means you cannot minimize the window and it will not show up on the taskbar
grab_anywhere	(bool) If True can use mouse to click and drag to move the window. Almost every location of the window will work except input fields on some systems
keep_on_top	(bool) If True, window will be created on top of all other windows on screen. It can be bumped down if another window created with this parm
resizable	(bool) If True, allows the user to resize the window. Note the not all Elements will change size or location when resizing.
disable_close	(bool) If True, the X button in the top right corner of the window will no work. Use with caution and always give a way out toyour users

Name	Meaning
disable_minimize	(bool) if True the user won't be able to minimize window. Good for taking over entire screen and staying that way.
right_click_menu	List[List[Union[List[str],str]]] A list of lists of Menu items to show when this element is right clicked. See user docs for exact format.
transparent_color	(str) Any portion of the window that has this color will be completely transparent. You can even click through these spots to the window under this window.
debugger_enabled	(bool) If True then the internal debugger will be enabled
finalize	(bool) If True then the Finalize method will be called. Use this rather than chaining .Finalize for cleaner code
element_justification	(str) All elements in the Window itself will have this justification 'left', 'right', 'center' are valid values
metadata	(Any) User metadata that can be set to ANYTHING

AddRow

Adds a single row of elements to a window's self.Rows variables. Generally speaking this is NOT how users should be building Window layouts. Users, create a single layout (a list of lists) and pass as a parameter to Window object, or call Window.Layout(layout)

```
AddRow(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Elements]

AddRows

Loops through a list of lists of elements and adds each row, list, to the layout. This is NOT the best way to go about creating a window. Sending the entire layout at one time and passing it as a parameter to the Window call is better.

AddRows(rows)

Parameter Descriptions:

Name	Meaning
rows	List[List[Elements]] A list of a list of elements

AlphaChannel

property: AlphaChannel

A property that changes the current alpha channel value (internal value)

Name	Meaning
return	(float) the current alpha channel setting according to self, not read directly from tkinter

BringToFront

Brings this window to the top of all other windows (perhaps may not be brought before a window made to "stay on top")

BringToFront()

Close

Closes window. Users can safely call even if window has been destroyed. Should always call when done with a window so that resources are properly freed up within your thread.

Close()

CurrentLocation

Get the current location of the window's top left corner

`CurrentLocation()`

Name	Meaning
return	Tuple[(int), (int)] The x and y location in tuple form (x,y)

Disable

Disables window from taking any input from the user

Disable()

DisableDebugger

Disable the internal debugger. By default the debugger is ENABLED

DisableDebugger()

Disappear

Causes a window to "disappear" from the screen, but remain on the taskbar. It does this by turning the alpha channel to 0. NOTE that on some platforms alpha is not supported. The window will remain showing on these platforms. The Raspberry Pi for example does not have an alpha setting

Disappear()

Elem

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

Elem(key, silent_on_error=False)

Parameter Descriptions:

Name	Meaning

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

Element

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
Element(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors

Name	Meaning
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

Enable

Re-enables window to take user input after having it be Disabled previously

```
Enable()
```

EnableDebugger

Enables the internal debugger. By default, the debugger IS enabled

```
EnableDebugger()
```

Fill

Fill in elements that are input fields with data based on a 'values dictionary'

```
Fill(values_dict)
```

Parameter Descriptions:

Name	Meaning
values_dict	(Dict[Any:Any]) {Element key : value} pairs
return	(Window) returns self so can be chained with other methods

Finalize

Use this method to cause your layout to built into a real tkinter window. In reality this method is like Read(timeout=0). It doesn't block and uses your layout to create tkinter widgets to represent the elements. Lots of action!

`Finalize()`

Name	Meaning
return	(Window) Returns 'self' so that method "Chaining" can happen (read up about it as it's very cool!)

Find

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
Find(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

FindElement

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
FindElement(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

FindElementWithFocus

Returns the Element that currently has focus as reported by tkinter. If no element is found None is returned!

`FindElementWithFocus()`

Name	Meaning
return	Union[Element, None] An Element if one has been found with focus or None if no element found

GetScreenDimensions

Get the screen dimensions. NOTE - you must have a window already open for this to work (blame tkinter not me)

`GetScreenDimensions()`

Name	Meaning
return	Union[Tuple[None, None], Tuple[width, height]] Tuple containing width and height of screen in pixels

GrabAnyWhereOff

Turns off Grab Anywhere functionality AFTER a window has been created. Don't try on a window that's not yet been Finalized or Read.

```
GrabAnyWhereOff()
```

GrabAnyWhereOn

Turns on Grab Anywhere functionality AFTER a window has been created. Don't try on a window that's not yet been Finalized or Read.

```
GrabAnyWhereOn()
```

Hide

Hides the window from the screen and the task bar

```
Hide()
```

Layout

Second of two preferred ways of telling a Window what its layout is. The other way is to pass the layout as a parameter to Window object. The parameter method is the currently preferred method. This call to Layout has been removed from examples contained in documents and in the Demo Programs. Trying to remove this call from history and replace with sending as a parameter to Window.

```
Layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Elements]] Your entire layout

Name	Meaning
return	(Window} self so that you can chain method calls

LoadFromDisk

Restore values from a previous call to SaveToDisk which saves the returned values dictionary in Pickle format

```
LoadFromDisk(filename)
```

Parameter Descriptions:

Name	Meaning
filename	(str) Pickle Filename to load

Maximize

Maximize the window. This is done differently on a windows system versus a linux or mac one. For non-Windows the root attribute '-fullscreen' is set to True. For Windows the "root" state is changed to "zoomed" The reason for the difference is the title bar is removed in some cases when using fullscreen option

```
Maximize()
```

Minimize

Minimize this window to the task bar

```
Minimize()
```

Move

Move the upper left corner of this window to the x,y coordinates provided

```
Move(x, y)
```

Parameter Descriptions:

Name	Meaning

Name	Meaning
x	(int) x coordinate in pixels
y	(int) y coordinate in pixels

Normal

Restore a window to a non-maximized state. Does different things depending on platform. See Maximize for more.

```
Normal()
```

Read

THE biggest deal method in the Window class! This is how you get all of your data from your Window. Pass in a timeout (in milliseconds) to wait for a maximum of timeout milliseconds. Will return timeout_key if no other GUI events happen first.

```
Read(timeout=None, timeout_key="__TIMEOUT__")
```

Parameter Descriptions:

Name	Meaning
timeout	(int) Milliseconds to wait until the Read will return IF no other GUI events happen first
timeout_key	(Any) The value that will be returned from the call if the timer expired
return	Tuple[(Any), Union[Dict[Any:Any]], List[Any], None] (event, values) (event or timeout_key or None, Dictionary of values or List of values from all elements in the Window)

Reappear

Causes a window previously made to "Disappear" (using that method). Does this by restoring the alpha channel

```
Reappear()
```

Refresh

Refreshes the window by calling tkroot.update(). Can sometimes get away with a refresh instead of a Read. Use this call when you want something to appear in your Window immediately (as soon as this function is called). Without this call your changes to a Window will not be visible to the user until the next Read call

`Refresh()`

Name	Meaning
return	(Window) `self` so that method calls can be easily "chained"

SaveToDisk

Saves the values contained in each of the input areas of the form. Basically saves what would be returned from a call to Read. It takes these results and saves them to disk using pickle

```
SaveToDisk(filename)
```

Parameter Descriptions:

Name	Meaning
filename	(str) Filename to save the values to in pickled form

SetAlpha

Sets the Alpha Channel for a window. Values are between 0 and 1 where 0 is completely transparent

```
SetAlpha(alpha)
```

Parameter Descriptions:

Name	Meaning
alpha	(float) 0 to 1. 0 is completely transparent. 1 is completely visible and solid (can't see through)

SetIcon

Sets the icon that is shown on the title bar and on the task bar. Can pass in: * a filename which must be a .ICO icon file for windows * a bytes object * a BASE64 encoded file held in a variable

SetIcon(icon=None, pngbase64=None)

Parameter Descriptions:

Name	Meaning
icon	(str) Filename or bytes object
pngbase64	(str) Base64 encoded GIF or PNG file

SetTransparentColor

Set the color that will be transparent in your window. Areas with this color will be SEE THROUGH.

SetTransparentColor(color)

Parameter Descriptions:

Name	Meaning
color	(str) Color string that defines the transparent color

Size

property: Size

Return the current size of the window in pixels

Name	Meaning
return	Tuple[(int), (int)] the (width, height) of the window

UnHide

Used to bring back a window that was previously hidden using the Hide method

UnHide()

VisibilityChanged

This is a completely dummy method that does nothing. It is here so that PySimpleGUIQt programs can make this call and then have that same source run on plain PySimpleGUI.

`VisibilityChanged()`

Name	Meaning
return	

add_row

Adds a single row of elements to a window's self.Rows variables. Generally speaking this is NOT how users should be building Window layouts. Users, create a single layout (a list of lists) and pass as a parameter to Window object, or call Window.Layout(layout)

```
add_row(args)
```

Parameter Descriptions:

Name	Meaning
*args	List[Elements]

add_rows

Loops through a list of lists of elements and adds each row, list, to the layout. This is NOT the best way to go about creating a window. Sending the entire layout at one time and passing it as a parameter to the Window call is better.

```
add_rows(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Elements]] A list of a list of elements

alpha_channel

property: alpha_channel

A property that changes the current alpha channel value (internal value)

Name	Meaning
return	(float) the current alpha channel setting according to self, not read directly from tkinter

bring_to_front

Brings this window to the top of all other windows (perhaps may not be brought before a window made to "stay on top")

bring_to_front()

close

Closes window. Users can safely call even if window has been destroyed. Should always call when done with a window so that resources are properly freed up within your thread.

close()

current_location

Get the current location of the window's top left corner

`current_location()`

Name	Meaning
return	Tuple[(int), (int)] The x and y location in tuple form (x,y)

disable

Disables window from taking any input from the user

disable()

disable_debugger

Disable the internal debugger. By default the debugger is ENABLED

disable_debugger()

disappear

Causes a window to "disappear" from the screen, but remain on the taskbar. It does this by turning the alpha channel to 0. NOTE that on some platforms alpha is not supported. The window will remain showing on these platforms. The Raspberry Pi for example does not have an alpha setting

```
disappear()
```

elem

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
elem(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

element

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
element(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

enable

Re-enables window to take user input after having it be Disabled previously

```
enable()
```

enable_debugger

Enables the internal debugger. By default, the debugger IS enabled

```
enable_debugger()
```

fill

Fill in elements that are input fields with data based on a 'values dictionary'

```
fill(values_dict)
```

Parameter Descriptions:

Name	Meaning
values_dict	(Dict[Any:Any]) {Element key : value} pairs
return	(Window) returns self so can be chained with other methods

finalize

Use this method to cause your layout to built into a real tkinter window. In reality this method is like Read(timeout=0). It doesn't block and uses your layout to create tkinter widgets to represent the elements. Lots of action!

`finalize()`

Name	Meaning
return	(Window) Returns 'self' so that method "Chaining" can happen (read up about it as it's very cool!)

find

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
find(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

find_element

Find element object associated with the provided key. THIS METHOD IS NO LONGER NEEDED to be called by the user

You can perform the same operation by writing this statement: element = window[key]

You can drop the entire "FindElement" function name and use [ ] instead.

Typically used in combination with a call to element's Update method (or any other element method!): window[key].Update(new_value)

Versus the "old way" window.FindElement(key).Update(new_value)

This call can be abbreviated to any of these: FindElement == Element == Find Rememeber that this call will return None if no match is found which may cause your code to crash if not checked for.

```
find_element(key, silent_on_error=False)
```

Parameter Descriptions:

Name	Meaning
key	(Any) Used with window.FindElement and with return values to uniquely identify this element
silent_on_error	(bool) If True do not display popup nor print warning of key errors

Name	Meaning
return	Union[Element, Error Element, None] Return value can be: * the Element that matches the supplied key if found * an Error Element if silent_on_error is False * None if silent_on_error True

find_element_with_focus

Returns the Element that currently has focus as reported by tkinter. If no element is found None is returned!

`find_element_with_focus()`

Name	Meaning
return	Union[Element, None] An Element if one has been found with focus or None if no element found

get_screen_dimensions

Get the screen dimensions. NOTE - you must have a window already open for this to work (blame tkinter not me)

`get_screen_dimensions()`

Name	Meaning
return	Union[Tuple[None, None], Tuple[width, height]] Tuple containing width and height of screen in pixels

get_screen_size

Returns the size of the "screen" as determined by tkinter. This can vary depending on your operating system and the number of monitors installed on your system. For Windows, the primary monitor's size is returns. On some multi-monitored Linux systems, the monitors are combined and the total size is reported as if one screen.

get_screen_size() -> Tuple[int, int] - Size of the screen in pixels as determined by tkinter

grab_any_where_off

Turns off Grab Anywhere functionality AFTER a window has been created. Don't try on a window that's not yet been Finalized or Read.

```
grab_any_where_off()
```

grab_any_where_on

Turns on Grab Anywhere functionality AFTER a window has been created. Don't try on a window that's not yet been Finalized or Read.

```
grab_any_where_on()
```

hide

Hides the window from the screen and the task bar

```
hide()
```

layout

Second of two preferred ways of telling a Window what its layout is. The other way is to pass the layout as a parameter to Window object. The parameter method is the currently preferred method. This call to Layout has been removed from examples contained in documents and in the Demo Programs. Trying to remove this call from history and replace with sending as a parameter to Window.

```
layout(rows)
```

Parameter Descriptions:

Name	Meaning
rows	List[List[Elements]] Your entire layout
return	(Window} self so that you can chain method calls

load_from_disk

Restore values from a previous call to SaveToDisk which saves the returned values dictionary in Pickle format

```
load_from_disk(filename)
```

Parameter Descriptions:

Name	Meaning
filename	(str) Pickle Filename to load

maximize

Maximize the window. This is done differently on a windows system versus a linux or mac one. For non-Windows the root attribute '-fullscreen' is set to True. For Windows the "root" state is changed to "zoomed" The reason for the difference is the title bar is removed in some cases when using fullscreen option

```
maximize()
```

minimize

Minimize this window to the task bar

```
minimize()
```

move

Move the upper left corner of this window to the x,y coordinates provided

```
move(x, y)
```

Parameter Descriptions:

Name	Meaning
x	(int) x coordinate in pixels
y	(int) y coordinate in pixels

normal

Restore a window to a non-maximized state. Does different things depending on platform. See Maximize for more.

```
normal()
```

read

THE biggest deal method in the Window class! This is how you get all of your data from your Window. Pass in a timeout (in milliseconds) to wait for a maximum of timeout milliseconds. Will return timeout_key if no other GUI events happen first.

```
read(timeout=None, timeout_key="__TIMEOUT__")
```

Parameter Descriptions:

Name	Meaning
timeout	(int) Milliseconds to wait until the Read will return IF no other GUI events happen first
timeout_key	(Any) The value that will be returned from the call if the timer expired
return	Tuple[(Any), Union[Dict[Any:Any]], List[Any], None] (event, values) (event or timeout_key or None, Dictionary of values or List of values from all elements in the Window)

reappear

Causes a window previously made to "Disappear" (using that method). Does this by restoring the alpha channel

```
reappear()
```

refresh

Refreshes the window by calling tkroot.update(). Can sometimes get away with a refresh instead of a Read. Use this call when you want something to appear in your Window immediately (as soon as this function is called). Without this call your changes to a Window will not be visible to the user until the next Read call

`refresh()`

Name	Meaning
return	(Window) `self` so that method calls can be easily "chained"

save_to_disk

Saves the values contained in each of the input areas of the form. Basically saves what would be returned from a call to Read. It takes these results and saves them to disk using pickle

save_to_disk(filename)

Parameter Descriptions:

Name	Meaning
filename	(str) Filename to save the values to in pickled form

set_alpha

Sets the Alpha Channel for a window. Values are between 0 and 1 where 0 is completely transparent

set_alpha(alpha)

Parameter Descriptions:

Name	Meaning
alpha	(float) 0 to 1. 0 is completely transparent. 1 is completely visible and solid (can't see through)

set_icon

Sets the icon that is shown on the title bar and on the task bar. Can pass in: * a filename which must be a .ICO icon file for windows * a bytes object * a BASE64 encoded file held in a variable

set_icon(icon=None, pngbase64=None)

Parameter Descriptions:

Name	Meaning
icon	(str) Filename or bytes object
pngbase64	(str) Base64 encoded GIF or PNG file

set_transparent_color

Set the color that will be transparent in your window. Areas with this color will be SEE THROUGH.

set_transparent_color(color)

Parameter Descriptions:

Name	Meaning
color	(str) Color string that defines the transparent color

size

property: size

Return the current size of the window in pixels

Name	Meaning
return	Tuple[(int), (int)] the (width, height) of the window

un_hide

Used to bring back a window that was previously hidden using the Hide method

```
un_hide()
```

visibility_changed

This is a completely dummy method that does nothing. It is here so that PySimpleGUIQt programs can make this call and then have that same source run on plain PySimpleGUI.

`visibility_changed()`

Name	Meaning
return	

```
CButton(button_text,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    bind_return_key=False,
    disabled=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	size of button image in pixels
image_subsample	amount to reduce the size of the image
border_width	width of border around element
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
font	specifies the font family, size, etc
bind_return_key	(Default = False)
disabled	set disable state for element (Default = False)

Name	Meaning
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
CalendarButton(button_text,
    target=(None, None),
    close_when_date_chosen=True,
    default_date_m_d_y=(None, None, None),
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    tooltip=None,
    border_width=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    locale=None,
    format=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
target	
close_when_date_chosen	(Default = True)
default_date_m_d_y	(Default = (None))
None	

Name	Meaning
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	(Default = (None))
image_subsample	amount to reduce the size of the image
tooltip	(str) text, that will appear when mouse hovers over the element
border_width	width of border around element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
locale	
format	
return	(Button)

```
Cancel(button_text="Cancel",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    tooltip=None,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Cancel')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

Change the "color scheme" of all future PySimpleGUI Windows. The scheme are string names that specify a group of colors. Background colors, text colors, button colors. There are 13 different color settings that are changed at one time using a single call to ChangeLookAndFeel The look and feel table itself has these indexe into the dictionary LOOK_AND_FEEL_TABLE SystemDefault Material1 Material2 Reddit Topanga GreenTan Dark LightGreen Dark2 Black Tan TanBlue DarkTanBlue DarkAmber DarkBlue Reds Green BluePurple Purple BlueMono GreenMono BrownBlue BrightColors NeutralBlue Kayak SandyBeach TealMono

```
ChangeLookAndFeel(index, force=False)
```

Parameter Descriptions:

Name	Meaning
index	(str) the name of the index into the Look and Feel table
force	(bool) if True allows Macs to use the look and feel feature. Otherwise Macs are blocked due to problems with button colors

```
CloseButton(button_text,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    bind_return_key=False,
    disabled=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	size of button image in pixels
image_subsample	amount to reduce the size of the image
border_width	width of border around element
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))

Name	Meaning
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
font	specifies the font family, size, etc
bind_return_key	(Default = False)
disabled	set disable state for element (Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
ColorChooserButton(button_text,
    target=(None, None),
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    tooltip=None,
    border_width=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
target	

Name	Meaning
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	(Default = (None))
image_subsample	amount to reduce the size of the image
tooltip	(str) text, that will appear when mouse hovers over the element
border_width	width of border around element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
Debug(button_text="",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    font=None,
    tooltip=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = '')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
font	specifies the font family, size, etc
tooltip	(str) text, that will appear when mouse hovers over the element
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
DummyButton(button_text,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    disabled=False,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	size of button image in pixels
image_subsample	amount to reduce the size of the image
border_width	width of border around element
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
font	specifies the font family, size, etc
disabled	set disable state for element (Default = False)
bind_return_key	(Default = False)

Name	Meaning
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
EasyPrint(args,
    size=(None, None),
    end=None,
    sep=None,
    location=(None, None),
    font=None,
    no_titlebar=False,
    no_button=False,
    grab_anywhere=False,
    keep_on_top=False,
    do_not_reroute_stdout=True)
```

Parameter Descriptions:

Name	Meaning
*args	
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
end	
sep	
location	Location on screen to display
font	specifies the font family, size, etc
no_titlebar	(Default = False)
no_button	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)

Name	Meaning
do_not_reroute_stdout	(Default = True)

```
EasyPrintClose()
```

```
Exit(button_text="Exit",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    tooltip=None,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Exit')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

```
FileBrowse(button_text="Browse",
    target=(555666777, -1),
    file_types=(('ALL Files', '*.*'),),
    initial_folder=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    change_submits=False,
    enable_events=False,
    font=None,
    disabled=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Browse')
target	key or (row,col) target for the button (Default value = (ThisRow, -1))
file_types	(Default value = (("ALL Files", ".")))
initial_folder	starting path for folders and files
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
change_submits	If True, pressing Enter key submits window (Default = False)
enable_events	Turns on the element specific events.(Default = False)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
disabled	set disable state for element (Default = False)
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

Name	Meaning
return	(Button)

```
FileSaveAs(button_text="Save As...",
    target=(555666777, -1),
    file_types=(('ALL Files', '*.*'),),
    initial_folder=None,
    disabled=False,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    change_submits=False,
    enable_events=False,
    font=None,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Save As...')
target	key or (row,col) target for the button (Default value = (ThisRow, -1))
file_types	(Default value = (("ALL Files", ".")))
initial_folder	starting path for folders and files
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
change_submits	If True, pressing Enter key submits window (Default = False)
enable_events	Turns on the element specific events.(Default = False)

Name	Meaning
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

return	(Button)

```
FilesBrowse(button_text="Browse",
    target=(555666777, -1),
    file_types=(('ALL Files', '*.*'),),
    disabled=False,
    initial_folder=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    change_submits=False,
    enable_events=False,
    font=None,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Browse')
target	key or (row,col) target for the button (Default value = (ThisRow, -1))
file_types	(Default value = (("ALL Files", ".")))
disabled	set disable state for element (Default = False)
initial_folder	starting path for folders and files
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text

Name	Meaning
button_color	button color (foreground, background)
change_submits	If True, pressing Enter key submits window (Default = False)
enable_events	Turns on the element specific events.(Default = False)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

Fills a window with values provided in a values dictionary { element_key : new_value }

```
FillFormWithValues(window, values_dict)
```

Parameter Descriptions:

Name	Meaning
window	(Window) The window object to fill
values_dict	(Dict[Any:Any]) A dictionary with element keys as key and value is values parm for Update call

```
FolderBrowse(button_text="Browse",
    target=(555666777, -1),
    initial_folder=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    change_submits=False,
    enable_events=False,
    font=None,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Browse')
target	key or (row,col) target for the button (Default value = (ThisRow, -1))
initial_folder	starting path for folders and files
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
change_submits	If True, pressing Enter key submits window (Default = False)
enable_events	Turns on the element specific events.(Default = False)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
Help(button_text="Help",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    font=None,
    tooltip=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Help')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
font	specifies the font family, size, etc
tooltip	(str) text, that will appear when mouse hovers over the element
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
ListOfLookAndFeelValues()
```

```
No(button_text="No",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    tooltip=None,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'No')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

```
OK(button_text="OK",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    bind_return_key=True,
    tooltip=None,
    font=None,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'OK')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text

Name	Meaning
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
bind_return_key	(Default = True)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
ObjToString(obj, extra="    ")
```

Parameter Descriptions:

Name	Meaning
obj	
extra	(Default value = ' ')

```
ObjToStringSingleObj(obj)
```

Parameter Descriptions:

Name	Meaning
obj	

```
Ok(button_text="Ok",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    bind_return_key=True,
    tooltip=None,
    font=None,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Ok')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
bind_return_key	(Default = True)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
OneLineProgressMeter(title,
    current_value,
    max_value,
    key,
    args,
    orientation="v",
    bar_color=(None, None),
    button_color=None,
    size=(20, 20),
    border_width=None,
    grab_anywhere=False)
```

Parameter Descriptions:

Name	Meaning
title	text to display
current_value	current progressbar value
max_value	max value of progressbar
key	Used with window.FindElement and with return values to uniquely identify this element
*args	stuff to output.
orientation	'horizontal' or 'vertical' ('h' or 'v' work) (Default value = 'vertical') (Default value = 'v')
bar_color	
button_color	button color (foreground, background)
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high (Default value = DEFAULT_PROGRESS_BAR_SIZE)
border_width	width of border around element
grab_anywhere	If True can grab anywhere to move the window (Default = False)

```
OneLineProgressMeterCancel(key)
```

Parameter Descriptions:

Name	Meaning
key	Used with window.FindElement and with return values to uniquely identify this element

```
Open(button_text="Open",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    bind_return_key=True,
    tooltip=None,
    font=None,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Open')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
bind_return_key	(Default = True)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

Show Popup box that doesn't block and closes itself

```
PopupQuick(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=2,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = True)
auto_close_duration	(Default value = 2)
non_blocking	(Default = True)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)

Name	Meaning
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Show Popup window with no titlebar, doesn't block, and auto closes itself.

```
PopupQuickMessage(args,
    title=None,
    button_type=5,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=2,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=True,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_NO_BUTTONS)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = True)
auto_close_duration	(Default value = 2)
non_blocking	(Default = True)

Name	Meaning
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = True)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Show a scrolled Popup window containing the user's text that was supplied. Use with as many items to print as you want, just like a print statement.

```
PopupScrolled(args,
    title=None,
    button_color=None,
    yes_no=False,
    auto_close=False,
    auto_close_duration=None,
    size=(None, None),
    location=(None, None),
    non_blocking=False)
```

Parameter Descriptions:

Name	Meaning
*args	(Any) Variable number of items to display
title	(str) Title to display in the window.
button_color	Tuple[str, str] button color (foreground, background)
yes_no	(bool) If True, displays Yes and No buttons instead of Ok
auto_close	(bool) if True window will close itself
auto_close_duration	Union[int, float] Older versions only accept int. Time in seconds until window will close
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high

Name	Meaning
location	Tuple[int, int] Location on the screen to place the upper left corner of the window
non_blocking	(bool) if True the call will immediately return rather than waiting on user input

return	Union[str, None, TIMEOUT_KEY] Returns text of the button that was pressed. None will be returned if user closed window with X

Popup that closes itself after some time period

```
PopupTimed(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text

Name	Meaning
auto_close	(Default = True)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Display Popup with Yes and No buttons

```
PopupYesNo(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	

Name	Meaning
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	Location on screen to display
return	Union["Yes", "No", None]

```
Print(args,
    size=(None, None),
    end=None,
    sep=None,
    location=(None, None),
    font=None,
    no_titlebar=False,
    no_button=False,
    grab_anywhere=False,
    keep_on_top=False,
    do_not_reroute_stdout=True)
```

Parameter Descriptions:

Name	Meaning
*args	

Name	Meaning
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
end	
sep	
location	Location on screen to display
font	specifies the font family, size, etc
no_titlebar	(Default = False)
no_button	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
do_not_reroute_stdout	(Default = True)

```
PrintClose()
```

```
Quit(button_text="Quit",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    tooltip=None,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Quit')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)

Name	Meaning
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
RButton(button_text,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    bind_return_key=False,
    disabled=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button

Name	Meaning
image_size	size of button image in pixels
image_subsample	amount to reduce the size of the image
border_width	width of border around element
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
font	specifies the font family, size, etc
bind_return_key	(Default = False)
disabled	set disable state for element (Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

```
ReadButton(button_text,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    bind_return_key=False,
    disabled=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	size of button image in pixels
image_subsample	amount to reduce the size of the image
border_width	width of border around element
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
font	specifies the font family, size, etc
bind_return_key	(Default = False)
disabled	set disable state for element (Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

```
RealtimeButton(button_text,
    image_filename=None,
    image_data=None,
    image_size=(None, None),
    image_subsample=None,
    border_width=None,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    font=None,
    disabled=False,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button
image_filename	image filename if there is a button image
image_data	in-RAM image to be displayed on button
image_size	size of button image in pixels
image_subsample	amount to reduce the size of the image
border_width	width of border around element
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high (Default = (None))
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
font	specifies the font family, size, etc
disabled	set disable state for element (Default = False)
bind_return_key	(Default = False)

Name	Meaning
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element

```
Save(button_text="Save",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    bind_return_key=True,
    disabled=False,
    tooltip=None,
    font=None,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Save')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
bind_return_key	(Default = True)
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
focus	if focus should be set to this
pad	Amount of padding to put around element

Name	Meaning
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

```
SaveAs(button_text="Save As...",
    target=(555666777, -1),
    file_types=(('ALL Files', '*.*'),),
    initial_folder=None,
    disabled=False,
    tooltip=None,
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    change_submits=False,
    enable_events=False,
    font=None,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Save As...')
target	key or (row,col) target for the button (Default value = (ThisRow, -1))
file_types	(Default value = (("ALL Files", ".")))
initial_folder	starting path for folders and files
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
change_submits	If True, pressing Enter key submits window (Default = False)

Name	Meaning
enable_events	Turns on the element specific events.(Default = False)
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

Show a scrolled Popup window containing the user's text that was supplied. Use with as many items to print as you want, just like a print statement.

```
ScrolledTextBox(args,
    title=None,
    button_color=None,
    yes_no=False,
    auto_close=False,
    auto_close_duration=None,
    size=(None, None),
    location=(None, None),
    non_blocking=False)
```

Parameter Descriptions:

Name	Meaning
*args	(Any) Variable number of items to display
title	(str) Title to display in the window.
button_color	Tuple[str, str] button color (foreground, background)
yes_no	(bool) If True, displays Yes and No buttons instead of Ok
auto_close	(bool) if True window will close itself
auto_close_duration	Union[int, float] Older versions only accept int. Time in seconds until window will close
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high

Name	Meaning
location	Tuple[int, int] Location on the screen to place the upper left corner of the window
non_blocking	(bool) if True the call will immediately return rather than waiting on user input
return	Union[str, None, TIMEOUT_KEY] Returns text of the button that was pressed. None will be returned if user closed window with X

Sets the icon which will be used any time a window is created if an icon is not provided when the window is created.

```
SetGlobalIcon(icon)
```

Parameter Descriptions:

Name	Meaning
icon	Union[bytes, str] Either a Base64 byte string or a filename

```
SetOptions(icon=None,
    button_color=None,
    element_size=(None, None),
    button_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,
    background_color=None,
    element_background_color=None,
    text_element_background_color=None,
    input_elements_background_color=None,
    input_text_color=None,
    scrollbar_color=None,
    text_color=None,
    element_text_color=None,
    debug_win_size=(None, None),
    window_location=(None, None),
    error_button_color=(None, None),
    tooltip_time=None)
```

Parameter Descriptions:

Name	Meaning
icon	filename of icon used for taskbar and title bar
button_color	button color (foreground, background)
element_size	Tuple[int, int] element size (width, height) in characters
button_element_size	Tuple[int, int]
margins	tkinter margins around outsize (Default = (None))

Name	Meaning
element_padding	(Default = (None))
auto_size_text	True if size should fit the text length
auto_size_buttons	
font	specifies the font family, size, etc
border_width	width of border around element
slider_border_width	
slider_relief	
slider_orientation	
autoclose_time	
message_box_line_width	
progress_meter_border_depth	
progress_meter_style	
progress_meter_relief	
progress_meter_color	
progress_meter_size	Tuple[int, int]
text_justification	
background_color	color of background
element_background_color	
text_element_background_color	
input_elements_background_color	
input_text_color	
scrollbar_color	
text_color	color of the text

Name	Meaning
element_text_color	
debug_win_size	Tuple[int, int] (Default = (None))
window_location	(Default = (None))
error_button_color	(Default = (None))
tooltip_time	time in milliseconds to wait before showing a tooltip. Default is 400ms

```
Submit(button_text="Submit",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    bind_return_key=True,
    tooltip=None,
    font=None,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Submit')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
bind_return_key	(Default = True)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
focus	if focus should be set to this

Name	Meaning
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

Time your code easily.... start the timer.

```
TimerStart()
```

Time your code easily.... stop the timer and print the number of ms since the timer start

```
TimerStop()
```

```
Yes(button_text="Yes",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    tooltip=None,
    font=None,
    bind_return_key=True,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Yes')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc

Name	Meaning
bind_return_key	(Default = True)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

Change the "color scheme" of all future PySimpleGUI Windows. The scheme are string names that specify a group of colors. Background colors, text colors, button colors. There are 13 different color settings that are changed at one time using a single call to ChangeLookAndFeel The look and feel table itself has these indexe into the dictionary LOOK_AND_FEEL_TABLE SystemDefault Material1 Material2 Reddit Topanga GreenTan Dark LightGreen Dark2 Black Tan TanBlue DarkTanBlue DarkAmber DarkBlue Reds Green BluePurple Purple BlueMono GreenMono BrownBlue BrightColors NeutralBlue Kayak SandyBeach TealMono

```
change_look_and_feel(index, force=False)
```

Parameter Descriptions:

Name	Meaning
index	(str) the name of the index into the Look and Feel table
force	(bool) if True allows Macs to use the look and feel feature. Otherwise Macs are blocked due to problems with button colors

```
easy_print(args,
    size=(None, None),
    end=None,
    sep=None,
    location=(None, None),
    font=None,
    no_titlebar=False,
    no_button=False,
    grab_anywhere=False,
    keep_on_top=False,
    do_not_reroute_stdout=True)
```

Parameter Descriptions:

Name	Meaning
*args	
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
end	
sep	
location	Location on screen to display
font	specifies the font family, size, etc
no_titlebar	(Default = False)
no_button	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
do_not_reroute_stdout	(Default = True)

```
easy_print_close()
```

```
eprint(args,
    size=(None, None),
    end=None,
    sep=None,
    location=(None, None),
    font=None,
    no_titlebar=False,
    no_button=False,
    grab_anywhere=False,
    keep_on_top=False,
    do_not_reroute_stdout=True)
```

Parameter Descriptions:

Name	Meaning
*args	
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
end	

Name	Meaning
sep	
location	Location on screen to display
font	specifies the font family, size, etc
no_titlebar	(Default = False)
no_button	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
do_not_reroute_stdout	(Default = True)

Fills a window with values provided in a values dictionary { element_key : new_value }

```
fill_form_with_values(window, values_dict)
```

Parameter Descriptions:

Name	Meaning
window	(Window) The window object to fill
values_dict	(Dict[Any:Any]) A dictionary with element keys as key and value is values parm for Update call

```
list_of_look_and_feel_values()
```

The PySimpleGUI "Test Harness". This is meant to be a super-quick test of the Elements.

```
main()
```

```
obj_to_string(obj, extra="    ")
```

Parameter Descriptions:

Name	Meaning
obj	
extra	(Default value = ' ')

```
obj_to_string_single_obj(obj)
```

Parameter Descriptions:

Name	Meaning
obj	

```
one_line_progress_meter(title,
    current_value,
    max_value,
    key,
    args,
    orientation="v",
    bar_color=(None, None),
    button_color=None,
    size=(20, 20),
    border_width=None,
    grab_anywhere=False)
```

Parameter Descriptions:

Name	Meaning
title	text to display
current_value	current progressbar value
max_value	max value of progressbar
key	Used with window.FindElement and with return values to uniquely identify this element
*args	stuff to output.
orientation	'horizontal' or 'vertical' ('h' or 'v' work) (Default value = 'vertical') (Default value = 'v')
bar_color	
button_color	button color (foreground, background)
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high (Default value = DEFAULT_PROGRESS_BAR_SIZE)
border_width	width of border around element

Name	Meaning
grab_anywhere	If True can grab anywhere to move the window (Default = False)

```
one_line_progress_meter_cancel(key)
```

Parameter Descriptions:

Name	Meaning
key	Used with window.FindElement and with return values to uniquely identify this element

Popup - Display a popup Window with as many parms as you wish to include. This is the GUI equivalent of the "print" statement. It's also great for "pausing" your program's flow until the user can read some error messages.

```
popup(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    button_type=0,
    auto_close=False,
    auto_close_duration=None,
    custom_text=(None, None),
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	(Any) Variable number of your arguments. Load up the call with stuff to see!
title	(str) Optional title for the window. If none provided, the first arg will be used instead.

Name	Meaning
button_color	Tuple[str, str] Color of the buttons shown (text color, button color)
background_color	(str) Window's background color
text_color	(str) text color
button_type	(enum) NOT USER SET! Determines which pre-defined buttons will be shown (Default value = POPUP_BUTTONS_OK). There are many Popup functions and they call Popup, changing this parameter to get the desired effect.
auto_close	(bool) If True the window will automatically close
auto_close_duration	(int) time in seconds to keep window open before closing it automatically
custom_text	Union[Tuple[str, str], str] A string or pair of strings that contain the text to display on the buttons
non_blocking	(bool) If True then will immediately return from the function without waiting for the user's input.
icon	Union[str, bytes] icon to display on the window. Same format as a Window call
line_width	(int) Width of lines in characters. Defaults to MESSAGE_BOX_LINE_WIDTH
font	Union[str, tuple(font name, size, modifiers) specifies the font family, size, etc
no_titlebar	(bool) If True will not show the frame around the window and the titlebar across the top
grab_anywhere	(bool) If True can grab anywhere to move the window. If no_titlebar is True, grab_anywhere should likely be enabled too
location	Tuple[int, int] Location on screen to display the top left corner of window. Defaults to window centered on screen

Name	Meaning
return	Union[str, None] Returns text of the button that was pressed. None will be returned if user closed window with X

Show animation one frame at a time. This function has its own internal clocking meaning you can call it at any frequency and the rate the frames of video is shown remains constant. Maybe your frames update every 30 ms but your event loop is running every 10 ms. You don't have to worry about delaying, just call it every time through the loop.

```
popup_animated(image_source,
    message=None,
    background_color=None,
    text_color=None,
    font=None,
    no_titlebar=True,
    grab_anywhere=True,
    keep_on_top=True,
    location=(None, None),
    alpha_channel=None,
    time_between_frames=0,
    transparent_color=None)
```

Parameter Descriptions:

Name	Meaning
image_source	Union[str, bytes] Either a filename or a base64 string.
message	(str) An optional message to be shown with the animation
background_color	(str) color of background
text_color	(str) color of the text
font	Union[str, tuple) specifies the font family, size, etc
no_titlebar	(bool) If True then the titlebar and window frame will not be shown
grab_anywhere	(bool) If True then you can move the window just clicking anywhere on window, hold and drag
keep_on_top	(bool) If True then Window will remain on top of all other windows currently shownn

Name	Meaning
location	(int, int) (x,y) location on the screen to place the top left corner of your window. Default is to center on screen
alpha_channel	(float) Window transparency 0 = invisible 1 = completely visible. Values between are see through
time_between_frames	(int) Amount of time in milliseconds between each frame
transparent_color	(str) This color will be completely see-through in your window. Can even click through

Display a Popup without a titlebar. Enables grab anywhere so you can move it

```
popup_annoying(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    grab_anywhere=True,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text

Name	Meaning
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
grab_anywhere	(Default = True)
location	

Popup that closes itself after some time period

```
popup_auto_close(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)

Name	Meaning
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = True)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Display Popup with "cancelled" button text

```
popup_cancel(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Popup with colored button and 'Error' as button text

```
popup_error(args,
    title=None,
    button_color=(None, None),
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	(Default = (None))

Display popup window with text entry field and browse button so that a file can be chosen by user.

```
popup_get_file(message,
    title=None,
    default_path="",
    default_extension="",
    save_as=False,
    multiple_files=False,
    file_types=(('ALL Files', '*.*'),),
    no_window=False,
    size=(None, None),
    button_color=None,
    background_color=None,
    text_color=None,
    icon=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None),
    initial_folder=None)
```

Parameter Descriptions:

Name	Meaning
message	(str) message displayed to user
title	(str) Window title
default_path	(str) path to display to user as starting point (filled into the input field)
default_extension	(str) If no extension entered by user, add this to filename (only used in saveas dialogs)
save_as	(bool) if True, the "save as" dialog is shown which will verify before overwriting
multiple_files	(bool) if True, then allows multiple files to be selected that are returned with ';' between each filename
file_types	Tuple[Tuple[str,str]] List of extensions to show using wildcards. All files (the default) = (("ALL Files", "."),)
no_window	(bool) if True, no PySimpleGUI window will be shown. Instead just the tkinter dialog is shown
size	Tuple[int, int] (width, height) of the InputText Element

Name	Meaning
button_color	Tuple[str, str] Color of the button (text, background)
background_color	(str) background color of the entire window
text_color	(str) color of the message text
icon	Union[bytes, str] filename or base64 string to be used for the window's icon
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
no_titlebar	(bool) If True no titlebar will be shown
grab_anywhere	(bool) If True can click and drag anywhere in the window to move the window
keep_on_top	(bool) If True the window will remain above all current windows
location	Tuyple[int, int] (x,y) Location on screen to display the upper left corner of window
initial_folder	(str) location in filesystem to begin browsing
return	Union[str, None] string representing the file(s) chosen, None if cancelled or window closed with X

Display popup with text entry field and browse button so that a folder can be chosen.

```
popup_get_folder(message,
    title=None,
    default_path="",
    no_window=False,
    size=(None, None),
    button_color=None,
    background_color=None,
    text_color=None,
    icon=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None),
    initial_folder=None)
```

Parameter Descriptions:

Name	Meaning
message	(str) message displayed to user
title	(str) Window title
default_path	(str) path to display to user as starting point (filled into the input field)
no_window	(bool) if True, no PySimpleGUI window will be shown. Instead just the tkinter dialog is shown
size	Tuple[int, int] (width, height) of the InputText Element
button_color	Tuple[str, str] Color of the button (text, background)
background_color	(str) background color of the entire window
text_color	(str) color of the message text
icon	Union[bytes, str] filename or base64 string to be used for the window's icon
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
no_titlebar	(bool) If True no titlebar will be shown
grab_anywhere	(bool) If True can click and drag anywhere in the window to move the window
keep_on_top	(bool) If True the window will remain above all current windows
location	Tuyple[int, int] (x,y) Location on screen to display the upper left corner of window
initial_folder	(str) location in filesystem to begin browsing
return	Union[str, None] string representing the path chosen, None if cancelled or window closed with X

Display Popup with text entry field. Returns the text entered or None if closed / cancelled

```
popup_get_text(message,
    title=None,
    default_text="",
    password_char="",
    size=(None, None),
    button_color=None,
    background_color=None,
    text_color=None,
    icon=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
message	(str) message displayed to user
title	(str) Window title
default_text	(str) default value to put into input area
password_char	(str) character to be shown instead of actually typed characters
size	Tuple[int, int] (width, height) of the InputText Element
button_color	Tuple[str, str] Color of the button (text, background)
background_color	(str) background color of the entire window
text_color	(str) color of the message text
icon	Union[bytes, str] filename or base64 string to be used for the window's icon
font	Union[str, Tuple[str, int]] specifies the font family, size, etc
no_titlebar	(bool) If True no titlebar will be shown
grab_anywhere	(bool) If True can click and drag anywhere in the window to move the window
keep_on_top	(bool) If True the window will remain above all current windows

Name	Meaning
location	Tuyple[int, int] (x,y) Location on screen to display the upper left corner of window
return	Union[str, None] Text entered or None if window was closed or cancel button clicked

Display a Popup without a titlebar. Enables grab anywhere so you can move it

```
popup_no_border(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    grab_anywhere=True,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	

Name	Meaning
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
grab_anywhere	(Default = True)
location	

Show a Popup but without any buttons

```
popup_no_buttons(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)

Name	Meaning
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Display a Popup without a titlebar. Enables grab anywhere so you can move it

```
popup_no_frame(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    grab_anywhere=True,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)

Name	Meaning
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
grab_anywhere	(Default = True)
location	

Display a Popup without a titlebar. Enables grab anywhere so you can move it

```
popup_no_titlebar(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    grab_anywhere=True,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	

Name	Meaning
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
grab_anywhere	(Default = True)
location	

Show Popup window and immediately return (does not block)

```
popup_no_wait(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = True)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Show Popup window and immediately return (does not block)

```
popup_non_blocking(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = True)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)

Name	Meaning
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Display Popup with OK button only

```
popup_ok(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display

Name	Meaning
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Show Popup box that doesn't block and closes itself

```
popup_quick(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=2,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text

Name	Meaning
auto_close	(Default = True)
auto_close_duration	(Default value = 2)
non_blocking	(Default = True)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Show Popup window with no titlebar, doesn't block, and auto closes itself.

```
popup_quick_message(args,
    title=None,
    button_type=5,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=2,
    non_blocking=True,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=True,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	

Name	Meaning
button_type	(Default value = POPUP_BUTTONS_NO_BUTTONS)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = True)
auto_close_duration	(Default value = 2)
non_blocking	(Default = True)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = True)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Show a scrolled Popup window containing the user's text that was supplied. Use with as many items to print as you want, just like a print statement.

```
popup_scrolled(args,
    title=None,
    button_color=None,
    yes_no=False,
    auto_close=False,
    auto_close_duration=None,
    size=(None, None),
    location=(None, None),
    non_blocking=False)
```

Parameter Descriptions:

Name	Meaning
*args	(Any) Variable number of items to display

Name	Meaning
title	(str) Title to display in the window.
button_color	Tuple[str, str] button color (foreground, background)
yes_no	(bool) If True, displays Yes and No buttons instead of Ok
auto_close	(bool) if True window will close itself
auto_close_duration	Union[int, float] Older versions only accept int. Time in seconds until window will close
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
location	Tuple[int, int] Location on the screen to place the upper left corner of the window
non_blocking	(bool) if True the call will immediately return rather than waiting on user input
return	Union[str, None, TIMEOUT_KEY] Returns text of the button that was pressed. None will be returned if user closed window with X

Popup that closes itself after some time period

```
popup_timed(args,
    title=None,
    button_type=0,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=True,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_type	(Default value = POPUP_BUTTONS_OK)
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = True)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	

Display Popup with Yes and No buttons

```
popup_yes_no(args,
    title=None,
    button_color=None,
    background_color=None,
    text_color=None,
    auto_close=False,
    auto_close_duration=None,
    non_blocking=False,
    icon=None,
    line_width=None,
    font=None,
    no_titlebar=False,
    grab_anywhere=False,
    keep_on_top=False,
    location=(None, None))
```

Parameter Descriptions:

Name	Meaning
*args	
title	
button_color	button color (foreground, background)
background_color	color of background
text_color	color of the text
auto_close	(Default = False)
auto_close_duration	
non_blocking	(Default = False)
icon	Icon to display
line_width	Width of lines in characters
font	specifies the font family, size, etc
no_titlebar	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
location	Location on screen to display

Name	Meaning
return	Union["Yes", "No", None]

```
quit(button_text="Quit",
    size=(None, None),
    auto_size_button=None,
    button_color=None,
    disabled=False,
    tooltip=None,
    font=None,
    bind_return_key=False,
    focus=False,
    pad=None,
    key=None,
    metadata=None)
```

Parameter Descriptions:

Name	Meaning
button_text	text in the button (Default value = 'Quit')
size	(w,h) w=characters-wide, h=rows-high
auto_size_button	True if button size is determined by button text
button_color	button color (foreground, background)
disabled	set disable state for element (Default = False)
tooltip	(str) text, that will appear when mouse hovers over the element
font	specifies the font family, size, etc
bind_return_key	(Default = False)
focus	if focus should be set to this
pad	Amount of padding to put around element
key	Used with window.FindElement and with return values to uniquely identify this element
return	(Button)

Sets the icon which will be used any time a window is created if an icon is not provided when the window is created.

```
set_global_icon(icon)
```

Parameter Descriptions:

Name	Meaning
icon	Union[bytes, str] Either a Base64 byte string or a filename

```
set_options(icon=None,
    button_color=None,
    element_size=(None, None),
    button_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,
    background_color=None,
    element_background_color=None,
    text_element_background_color=None,
    input_elements_background_color=None,
    input_text_color=None,
    scrollbar_color=None,
    text_color=None,
    element_text_color=None,
    debug_win_size=(None, None),
    window_location=(None, None),
    error_button_color=(None, None),
    tooltip_time=None)
```

Parameter Descriptions:

Name	Meaning
icon	filename of icon used for taskbar and title bar
button_color	button color (foreground, background)
element_size	Tuple[int, int] element size (width, height) in characters
button_element_size	Tuple[int, int]
margins	tkinter margins around outsize (Default = (None))
element_padding	(Default = (None))
auto_size_text	True if size should fit the text length
auto_size_buttons	
font	specifies the font family, size, etc
border_width	width of border around element
slider_border_width	
slider_relief	
slider_orientation	
autoclose_time	
message_box_line_width	
progress_meter_border_depth	
progress_meter_style	
progress_meter_relief	
progress_meter_color	
progress_meter_size	Tuple[int, int]
text_justification	
background_color	color of background

Name	Meaning
element_background_color	
text_element_background_color	
input_elements_background_color	
input_text_color	
scrollbar_color	
text_color	color of the text
element_text_color	
debug_win_size	Tuple[int, int] (Default = (None))
window_location	(Default = (None))
error_button_color	(Default = (None))
tooltip_time	time in milliseconds to wait before showing a tooltip. Default is 400ms

```
sgprint(args,
    size=(None, None),
    end=None,
    sep=None,
    location=(None, None),
    font=None,
    no_titlebar=False,
    no_button=False,
    grab_anywhere=False,
    keep_on_top=False,
    do_not_reroute_stdout=True)
```

Parameter Descriptions:

Name	Meaning
*args	
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
end	

Name	Meaning
sep	
location	Location on screen to display
font	specifies the font family, size, etc
no_titlebar	(Default = False)
no_button	(Default = False)
grab_anywhere	If True can grab anywhere to move the window (Default = False)
do_not_reroute_stdout	(Default = True)

```
sgprint_close()
```

Shows the smaller "popout" window. Default location is the upper right corner of your screen

```
show_debugger_popout_window(location=(None, None), args)
```

Parameter Descriptions:

Name	Meaning
location	Tuple[int, int] Locations (x,y) on the screen to place upper left corner of the window
*args	Not used

Shows the large main debugger window

```
show_debugger_window(location=(None, None), args)
```

Parameter Descriptions:

Name	Meaning
location	Tuple[int, int] Locations (x,y) on the screen to place upper left corner of the window
*args	Not used

Show a scrolled Popup window containing the user's text that was supplied. Use with as many items to print as you want, just like a print statement.

```
sprint(args,
    title=None,
    button_color=None,
    yes_no=False,
    auto_close=False,
    auto_close_duration=None,
    size=(None, None),
    location=(None, None),
    non_blocking=False)
```

Parameter Descriptions:

Name	Meaning
*args	(Any) Variable number of items to display
title	(str) Title to display in the window.
button_color	Tuple[str, str] button color (foreground, background)
yes_no	(bool) If True, displays Yes and No buttons instead of Ok
auto_close	(bool) if True window will close itself
auto_close_duration	Union[int, float] Older versions only accept int. Time in seconds until window will close
size	Tuple[int, int] (w,h) w=characters-wide, h=rows-high
location	Tuple[int, int] Location on the screen to place the upper left corner of the window
non_blocking	(bool) if True the call will immediately return rather than waiting on user input
return	Union[str, None, TIMEOUT_KEY] Returns text of the button that was pressed. None will be returned if user closed window with X

The PySimpleGUI "Test Harness". This is meant to be a super-quick test of the Elements.

```
test()
```

"Demo Programs" Applications

There are too many to list!!

There are over 170 sample programs to give you a jump start.

These programs are an integral part of the overall PySimpleGUI documentation and learning system. They will give you a headstart in a way you can learn from and understand. They also show you integration techiques to other packages that have been figured out for you.

You will find Demo Programs located in a subfolder named "Demo Programs" under the top level and each of the PySimpleGUI ports on GitHub.

Demo programs for plain PySimpleGUI (tkinter) https://github.com/PySimpleGUI/PySimpleGUI/tree/master/DemoPrograms

Demo programs for PySimpleGUIQt: https://github.com/PySimpleGUI/PySimpleGUI/tree/master/PySimpleGUIQt/Demo%20Programs

Demo programs for PySimpleGUIWx: https://github.com/PySimpleGUI/PySimpleGUI/tree/master/PySimpleGUIWx/Demo%20Programs

Demo programs for PySimpleGUIWeb: https://github.com/PySimpleGUI/PySimpleGUI/tree/master/PySimpleGUIWeb/Demo%20Programs

There are not many programs under each of the port's folders because the main Demo Programs should run on all of the other platforms with minimal changes (often only the import statement changes).

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 * OpenCV * pymunk * psutil * pygame * Forecastio

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

Creating a Mac App File

There are reports that PyInstaller can be used to create App files. It's not been officially tested.

Run this command on your Mac

> pyinstaller --onefile --add-binary='/System/Library/Frameworks/Tk.framework/Tk':'tk' --add-binary='/System/Library/Frameworks/Tcl.framework/Tcl':'tcl' your_program.py

This info was located on Reddit with the source traced back to: https://github.com/pyinstaller/pyinstaller/issues/1350

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

You can change defaults and colors of a large number of things in PySimpleGUI quite easily.

`ChangeLookAndFeel`

Want a quick way of making your windows look a LOT better? Try calling `ChangeLookAndFeel`. It will, in a single call, set various color values to widgets, background, text, etc.

Or dial in the look and feel (and a whole lot more) 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.

```
sg.ChangeLookAndFeel('GreenTan')
```

Valid look and feel values are currently:

```
SystemDefault
Reddit
Topanga
GreenTan
Dark
LightGreen
Dark2
Black
Tan
TanBlue
DarkTanBlue
DarkAmber
DarkBlue
Reds
Green
BluePurple
Purple
BlueMono
GreenMono
BrownBlue
BrightColors
NeutralBlue
Kayak
SandyBeach
TealMono
```

The way this call actually works is that it calls `SetOptions` with a LOT of color settings. Here is the actual call that's made. As you can see lots of stuff is defined for you.

```
SetOptions(background_color=colors['BACKGROUND'],
          text_element_background_color=colors['BACKGROUND'],
          element_background_color=colors['BACKGROUND'],
          text_color=colors['TEXT'],
          input_elements_background_color=colors['INPUT'],
          button_color=colors['BUTTON'],
          progress_meter_color=colors['PROGRESS'],
          border_width=colors['BORDER'],
          slider_border_width=colors['SLIDER_DEPTH'],
          progress_meter_border_depth=colors['PROGRESS_DEPTH'],
          scrollbar_color=(colors['SCROLL']),
          element_text_color=colors['TEXT'],
          input_text_color=colors['TEXT_INPUT'])
```

To see the latest list of color choices you can call `ListOfLookAndFeelValues()`

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

Well, there are a few quirks, and problems of course. Check the GitHub Issues database for a list of them.

As previously mentioned this is also where you should post all problems and enhancements.

MACS + tkinter = SUCKS

Not sure why, but for over a year and a half, setting the color of buttons does not work on Macs. There have been numerous other problems. Checking the Issues database is the best place to see what they are. If there was a magic wand it would have been used long ago to fix these problems, but there does not appear to be a magic fix.

This was already mentioned at the top of this document but want to make sure it's covered as a "known issue"

Multiple threads

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

Tkinter also wants to be the MAIN thread in your code. So, if you have to run multiple threads, make sure the GUI is the main thread.

Other than that, feel free to use threads with PySimpleGUI on all of the ports. You'll find a good example for how to run "long running tasks" in your event loop by looking at the demo program: `Demo_Multithreaded_Long_Tasks.py`

Contributing

Core Code

Core code changes/pull requests are not being accepted at this time.

Demos

You're welcome to share a PySimpleGUI program you've written that you think fits the model of a PySimpleGUI Demo Program.

GitHub Repos

If you've created a GitHub for your project that uses PySimpleGUI then please submit it to be included in this document or on the PySimpleGUI GitHub site. Also, you'll find a lot more people will look at your code, explore your repo if you have posted **screen shots in your**

readme. People *love* success stories and showing your GUI's screen shows you've been successful. Everyone wins!

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

Version	Description
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
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

Version	Description
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.
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\
01.00.00 for 2.7	Sept 25, 2018 - First release for 2.7
03.08.04	Sept 30, 2018 - See release notes
03.09.00	Oct 1, 2018

Version	Description
2.7 01.01.00	Oct 1, 2018
2.7 01.01.02	Oct 8, 2018
03.09.01	Oct 8, 2018
3.9.3 & 1.1.3	Oct 11, 2018
3.9.4 & 1.1.4	Oct 16, 2018
3.10.1 & 1.2.1	Oct 20, 2018
3.10.3 & 1.2.3	Oct 23, 2018
3.11.0 & 1.11.0	Oct 28, 2018
3.12.0 & 1.12.0	Oct 28, 2018
3.13.0 & 1.13.0	Oct 29, 2018
3.14.0 & 1.14.0	Nov 2, 2018
3.15.0 & 1.15.0	Nov 20, 2018
3.16.0 & 1.16.0	Nov 26, 2018
3.17.0 & 1.17.0	Dec 1, 2018

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 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)

3.8.6

- Fix for Menus.
- Fixed table colors. Now they work
- Fixed returning keys for tabs
- Window Hide / UnHide methods
- Changed all Popups to remove context manager
- Error checking for Graphing objects and for Element Updates

3.9.0 & 1.1.0

- The FIRST UNIFIED version of the code!
- Python 2.7 got a TON of features . Look back to 1.0 release for the list
- Tab locations - Can place Tabs on top, bottom, left, right now instead of only the top

3.9.1 & 1.1.2

- Tab features
- Themes
- Enable / Disable
- Tab text colors
- Selected tab color
- New GetListValues method for Listbox
- Can now have multiple progress bars in 1 window

- Fix for closing debug-output window with other windows open
- Topanga Look and Feel setting
- User can create new look and feel settings / can access the look and feel table
- New PopupQuick call. Shows a non-blocking popup window with auto-close
- Tree Element partially done (don't use despite it showing up)

3.9.3 & 1.1.3

- Disabled setting when creating element for:
- Input
- Combo
- Option Menu
- Listbox
- Radio
- Checkbox
- Spinner
- Multiline
- Buttons
- Slider
- Doc strings on all Elements updated
- Buttons can take image data as well as image files
- Button Update can change images
- Images can have background color
- Table element new num_rows parameter
- Table Element new alternating_row_color parameter
- Tree Element
- Window Disappear / Reappear methods
- Popup buttons resized to same size
- Exposed look and feel table

3.9.4 & 1.1.4

- Parameter order change for Button.Update so that new button ext is at front
- New Graph.DrawArc method
- Slider tick interval parameter for labeling sliders
- Menu tearoff now disabled by default
- Tree Data printing simplified and made prettier
- Window resizable parameter. Defaults to not resizable
- Button images can have text over them now
- BUG fix in listbox double-click. First bug fix in months
- New Look And Feel capability. List predefined settings using ListOfLookAndFeelValues

3.10.1 & 1.2.1

- Combobox new readonly parameter in init and Update
- Better default sizes for Slider
- Read of Tables now returns which rows are selected (big damned deal feature)
- PARTIAL support of Table.Update with new values (use at your own peril)
- Alpha channel setting for Windows
- Timeout setting for Window.Read (big damned deal feature)
- Icon can be base64 image now in SetIcon call
- Window.FindElementWithFocus call
- Window.Move allows moving window anywhere on screen
- Window.Minimize will minimize to taskbar
- Button background color can be set to system default (i.e. not changed)

3.10.2 & 1.2.2

Emergency patch release... going out same day as previous release * The timeout timer for the new Read with timer wasn't being properly shut down * The Image.Update method appears to not have been written correctly. It didn't handle base64 images like the other elements that deal with images (buttons)

3.10.3 & 1.2.3

- New element - Vertical Separator
- New parameter for InputText - change_submits. If True will cause Read to return when a button fills in the InputText element
- Read with timeout = 0 is same as read non blocking and is the new preferred method
- Will return event == None if window closed
- New Close method will close all window types
- Scrollbars for Tables automatically added (no need for a Column Element)
- Table Update method complete
- Turned off expand when packing row frame... was accidentally turned on (primary reason for this release)
- Try added to Image Update so won't crash if bad image passed in

3.11.0 & 1.11.0

- Syncing up the second digit of the releases so that they stay in sync better. the 2.7 release is built literally from the 3.x code so they really are the same
- Reworked Read call... significantly.
- Realtime buttons work with timeouts or blocking read
- Removed default value parm on Buttons and Button Updates
- New Tree Element parm show_expanded. Causes Tree to be shown as fully expanded
- Tree Element now returns which rows are selected when Read
- New Window method BringToFront

- Shortcut buttons no longer close windows!
- Added CloseButton, CButton that closes the windows

3.12.0 & 1.12.0

- Changed Button to be the same as ReadButton which means it will no longer close the window
- All shortcut buttons no longer close the window
- Updating a table clears selected rows information in return values
- Progress meter uses new CloseButton
- Popups use new CloseButton

3.13.0 & 1.13.0

- Improved multiple window handling of Popups when the X is used to close
- Change submits added for:
- Multiline
- Input Text
- Table
- Tree
- Option to close calendar chooser when date selected
- Update for Tree Element
- Scroll bars for Trees

3.14.0 & 1.14.0

- More windowing changes...
    - using a hidden root windowing (Tk())
    - all children are Toplevel() windows
- Read only setting for:
    - Input Text
    - Multiline
- Font setting for InputCombo, Multiline
- change_submits setting for Radio Element
- SetFocus for multiline, input elements
- Default mon, day, year for calendar chooser button
- Tree element update, added ability to change a single key
- Message parm removed from ReadNonBlocking
- Fix for closing windows using X
- CurrentLocation method for Windows

- Debug Window options
  - location
  - font
  - no_button
  - no_titlebar
  - grab_anywhere
  - keep_on_top
- New Print / EasyPrint options
  - location
  - font
  - no_button
  - no_titlebar
  - grab_anywhere
  - keep_on_top
- New popup, PopupQuickMessage
- PopupGetFolder, PopupGetFile new initial_folder parm

3.15.0 & 1.15.0

- Error checking for InputText.Get method
- Text color, background color added to multiline element.Update
- Update method for Output Element - gives ability to clear the output
- Graph Element - Read returns values if new flages set
  - Change submits, drag submits
  - Returns x,y coordinates
- Column element new parm vertical_scroll_only
- Table element new parm - bind return key - returns if return or double click
- New Window parms - size, disable_close
- "Better" multiwindow capabilities
- Window.Size property
- Popups - new title parm, custom_text
  - title sets the window title
  - custom_text - single string or tuple string sets text on button(s)

3.16.0 & 1.16.0

- Bug fix in PopupScrolled
- New `Element` shortcut function for `FindElement`
- Dummy Stretch Element made for backwards compatibility with Qt
- Timer function prints in milliseconds now, was seconds

3.17.0 &1.17.0 2-Dec-2018

3.17.0 2-Dec-2017 * Tooltip offset now programmable. Set variable DEFAULT_TOOLTIP_OFFSET. Defaults to (20,-20) * Tooltips are always on top now * Disable menu items * Menu items can have keys * StatusBar Element (preparing for a real status bar in Qt) * enable_events parameter added to ALL Elements capable of generating events * InputText.Update select parameter will select the input text * Listbox.Update - set_to_index parameter will select a single items * Menus can be updated! * Menus have an entry in the return values * LayoutAndRead depricated * Multi-window support continues (X detection) * PopupScrolled now has a location parameter * row_height parameter to Table Element * Stretch Element (DUMMY) so that can be source code compatible with Qt * ButtonMenu Element (DUMMY) so can be source code compat with Qt. Will implement eventually

3.18.0 11-Dec-2018

NOTE - **Menus are broken** on version 2.7. Don't know how long they've been this way. Please get off legacy Python if that's what you're running.

- Default progress bar length changed to shorter
- Master window and tracking of num open windows moved from global to Window class variable
- Element visibility setting (when created and when Updating element)
- Input text visiblity
- Combo visiblity
- Combo replaces InputCombo as the primary class name
- Option menu visibility
- Listbox visiblity
- Listbox new SetFocus method
- Radio visibility
- Checkbox visibility
- Spin visiblity
- Spin new Get method returns current value
- Multiline visiblity
- Text visibility
- StatusBar visiblity
- Output visibility
- Button visibility
- Button SetFocus
- ProgressBar - New Update method (used only for visibility)
- Image - clickable images! enable_events parameter
- Image visibility
- Canvas visibility
- Graph visibility
- Graph - new DrawImage capability (finally)
- Frame visibility

- Tab visibility (may not be fully functional)
- TabGroup visibility
- Slider visibility
- Slider - new disable_number_display parameter
- Column visibilty
- Menu visibility - Not functional
- Table visibility
- Table - new num_rows parm for Update - changes number of visible rows
- Tree visiblity
- Window - New element_padding parameter will get padding for entire window
- OneLineProgressMeter - Completely REPLACED the implementation
- OneLineProgressMeter - can get reason for the cancellation (cancel button versus X)
- EasyProgressMeter - completely removed. Use OneLineProgressMeter instead
- Debug window, EasyPrint, Print - debug window will re-open if printed to after being closed
- SetOptions - can change the error button color
- Much bigger window created when running PySimpleGUI.py by itself. Meant to help with regression testing

3.19.2 13-Dec-2018

- Warning for Mac's when trying to change button color
- New parms for Button.Update - image_size and image_subsample
- Buttons - remove highlight when border depth == 0
- OneLineProgressMeter - better layout implementation

3.20.0 & 1.20.0 18-Dec-2018

- New Pane Element
- Graph.DeleteFigure method
- disable_minimize - New parameter for Window
- Fix for 2.7 menus
- Debug Window no longer re-routes stdout by default
- Can re-route by specifying in Print / EasyPrint call
- New non-blocking for PopupScrolled
- Can set title for PopupScrolled window

3.21.0 & 1.21.0 28-Dec-2018

- ButtonMenu Element
- Embedded base64 default icon
- Input Text Right click menu

- Disabled Input Text are now 'readonly' instead of disabled
- Listbox right click menu
- Multiline right click menu
- Text right click menu
- Output right click menu
- Image right click menu
- Canvas right click menu
- Graph right click menu
- Frame right click menu
- Tab, tabgroup right click menu (unsure if works correctly)
- Column right click menu
- Table right click menu
- Tree right click menu
- Window level right click menu
- Window icon can be filename or bytes (Base64 string)
- Window.Maximize method
- Attempted to use Styles better with Combobox
- Fixed bug blocking setting bar colors in OneLineProgressMeter

3.22.0 PySimpleGUI / 1.22.0 PySimpleGUI27

- Added type hints to some portions of the code
- Output element can be made invisible
- Image sizing and subsample for Button images
- Invisibility for ButtonMenusup
- Attempt at specifying size of Column elements (limited success)
- Table Element
- New row_colors parameter
- New vertical_scroll_only parameter - NOTE - will have to disable to get horizontal scrollbars
- Tree Element
- New row_height parameter
- New feature - Icons for tree entries using filename or Base64 images
- Fix for bug sending back continuous mouse events
- New parameter silence_on_error for FindElement / Element calls
- Slider returns float now
- Fix for Menus when using Python 2.7
- Combobox Styling (again)

3.2.0 PySimpleGUI / 1.23.0 PySimpleGUI27 16-Jan-2019

- Animated GIFs!

- Calendar Chooser stays on top of other windows
- Fixed bug of no column headings for Tables
- Tables now use the font parameter

3.24.0 1.24.0 16-Jan-2019

PopupAnimated - A popup call for showing "loading" type of windows

3.25 & 1.25 20-Feb-2019

- Comments :-)
- Convert Text to string right away
- Caught exceptions when main program shut down with X
- Caught exceptions in all of the graphics primitives
- Added parameter exportselection=False to Listbox so can use multiple listboxes
- OneLineProgressMeter - Can now change the text on every call if desired

3.27.0 PySimpleGUI 31-Mar-2019

Mixup.... 3.26 changes don't appear to have been correctly released so releasing in 3.27 now

- do_not_clear now defaults to TRUE!!!
- Input Element
- Multiline Element
- Enable Radio Buttons to be in different containers
- Ability to modify Autoscroll setting in Multiline.Update call
- PopupGetFolder, PopupGetFile, PopupGetText - title defaults to message if none provided
- PopupAnimated - image_source can be a filename or bytes (base64)
- Option Menu can now have values updated

3.28.0 11-Apr-2019 PySimpleGUI

- NEW Window Parameter - layout - second parameter. Can pass in layout directly now!
- New shortcuts
  - I = InputText
  - B = Btn = Butt = Button
- Convert button text to string when creating buttons
- Buttons are returned now as well as input fields when searching for element with focus

3.29 22-Apr-2019

- New method for `Graph` - `RelocateFigure`

- Output Element no longer accepts focus

3.32.0 PySimpleGUI 24-May-2019

- Rework of ALLL Tooltips. Was always displaying at uttuper left part of element. Not displays closer to where mouse entered or edited
- New Element.Widget base class variable. Brings tkinter into the newer architecture of user accessibility to underlying GUI Frameworks' widgets
- New SetTooltip Element method. Means all Elements gain this method. Can set the tooltip on the fly now for all elements
- Include scroll bar when making visible / invisible Listbox Elements
- New Radio Element method - `Radio.ResetGroup()` sets all elements in the Radio Group to False* Added borderwidth to Multiline Element
- `Button.Click()` - new method - Generates a button click even as if a user clicked a button (at the tkinter level)
- Made a Graph.Images dictionary to keep track of images being used in a graph. When graph is deleted, all of the accociated images should be deleted too.'
- Added `Graph.SetFocus()` to give a Graph Element the focus just as you can input elements
- Table new parameter - `hide_vertical_scroll` if True will hide the table's vertical bars
- Window - new parameter - `transparent_color`. Causes a single color to become completely transparent such that you see through the window, you can click through the window. Its like tineows never was there.
- The new `Window.AllKeysDict = {}` has been adopted by all PySimpleGUI ports. It's a new method of automatically creating missing keys, storing and retrieving keys in general for a window.
- Changed how `window.Maximize` is implemented previously used the '-fullscreen' attribute. Now uses the 'zoomed' state
- Window gets a new `Normal()` method to return from Maximize state. Sets root.state('normal')
- Window.Close() now closes the special `Window.hidden_master_root` window when the "last" window is closed
- `Window.SetTransparentColor` method added. Same effect as if window was created with parameter set
- An Element's Widget stored in `.Widget` attribute
- Making ComboBox's ID unique by using it's Key
- Changed Multiline to be sunken and have a border depth setting now
- Removed a second canvas that was being used for Graph element.
- Changed how no titlebar is implemented running on Linux versus Windows. -type splash now used for Linux
- PopupScrolled - Added back using CloseButton to close the window
- Fixed PopupGetFolder to use correct PySimpleGUI program constructs (keys)

- PopupGetText populated values carrectly using the value variable, used keys
- PopupAnimated finally gets a completely transparent background

3.33.0 and 1.33 PySimpleGUI 25-May-2019

Emergency fix due to debugger. Old bug was that Image Element was not testing for COLOR_SYSTEM_DEFAULT correctly.

3.34.0 PySimpleGUI & 1.34.0 PySimpleGUI27 25-May-2019

pip rhw w cenf * Fixed Window.Maximize and Window.Normal - needed special code for Linux * Check for DEFAULT_SCROLLBAR_COLOR not being the COLOR_SYSTEM_DEFAULT (crashed)

3.35 PySimpleGUI & 1.35 PySimpleGUI27 27-May-2019

Bug fix - when setting default for Checkbox it was also disabling the element!

3.36 PySimpleGUI & 1.36 PySimpleGUI27 29-May-2019

A combination of user requests, and needs of new `imwatchingyou` debugger

- New Debugger Icon for future built-in debugger
- Fixed bug in FindBoundReturnKey - needed to also check Panes
- NEW Window functions to turn on/off the Grab Anywhere feature
  - `Window.GrabAnyWhereOn()`
  - `Window.GrabAnyWhereOff()`
- New "Debugger" button that's built-in like other buttons. It's a TINY button with a logo. For future use when a debugger is built into PySimpleGUI itself (SOON!)
- Change Text Element Wrap Length calculation. Went fromn +40 pixels to +10 pixels in formula
- PopupGetFile has new parameter - `multiple_files` . If True then allows selection of multiple files

3.37 PySimpleGUI & 1.37 PySimpleGUI27 1-June-2019

- The built-in debugger is HERE - might not WORK exactly yet, but a lot of code went into te PySimpleGUI.py file for this. At the moment, the `imwatchingyou` package is THE way to use a PySimpleGUI debugger. But soon enough you won't need that project in order to debug your program.
- Some strange code reformatting snuck in. There are 351 differences between this and previous release. I'm not sure what happened but am looking at every change by hand.

- New Calendar Button features
    - locale, format - new parameters to TKCalendar call
    - Use custom icon for window if one has been set
    - New parameters to CalendarButton - `locale` , `format`
- The bulk of the built-in PySimpleGUI debugger has been added but is not yet "officially supported". Try pressing "break" or "ctrl+break" on your keyboard.
    - New bindings for break / pause button and debugger
    - New Debug button will launch debugger.
    - New parameter `debugger_enabled` added to Window call. Default is **enabled**.
    - Your progam's call to Read is all that's needed to refresh debugger
    - New `Window` methods to control debugger access
        - `EnableDebugger` - turns on HOTKEYS to debugger
        - `DisableDebugger` - turns off HOTKEYS to debugger
- Restored wrap len for Text elements back from +10 to +40 pixels
- `PopupGetFolder` , `PopupGetFile` - fixed so that the "hidden" master window stays hidden (a Linux problem)
- Added support for Multiple Files to `PopupGetFiles` when no_window option has been set.

3.38 PySimpleGUI, 1.38 PySimpleGUI27

- Multiline - now has a "read only" state if created as "Disabled"
- Multiline - If window is created as resizable, then Multiline Elements will now expand when the window is enlarged, a feature long asked for.
- Output Element expands in the Y Direction
- "Expandable Rows" option added to PackFormIntoFrame allowing future elements to also expand
- Error Element - silence_on_error option
- Text Element wrapping - FINALLY got it right? No more "Fudge factor" added
- PopupScrolled - Windows are now resizable
- Option to "launch built-in debugger" from the test harness
- Rememeber that the Debugger is still in this code! It may or may not be operational as it's one version back from the latest release of the `imwatchingyou` debugger code. This code needs to be integrated back in

3.39 PySimpleGUI & 1.39 PySimpleGUI27 13-June-2019

- Ported the imwatchingyou debugger code into PySimpleGUI code
    - Replaced old debugger built-in code with the newer imwatchingyou version
    - Required removing all of the 'sg.' before PySimpleGUI calls since not importing
    - Dynamically create the debugger object when first call to `refresh` or `show` is made

- Started the procecss of renaming Class Methods that are private to start with _
- Needed for the automatic documentation generation that's being worked on
- Fixed crash when clicking the Debug button
- Fixed bug in DeleteFigure. Needed to delete image separately
- Added more type hints
- New `TabGroup` method `SelectTab(index)` selects a `Tab` within a `TabGroup`
- New `Table.Update` parameter - `select_rows`. List of rows to select (0 is first)
- Error checking in `Window.Layout` provides error "hints" to the user
    - Looks for badly placed ']'
    - Looks for functions missing '()'
    - Pops up a window warning user instead of crashing
    - May have to revisit if the popups start getting in the way
- New implementations of `Window.Disable()` and `Window.Enable()`
    - Previously did not work correctly at all
    - Now using the "-disabled" attribute
- Allow Comboboxes to have empty starting values
    - Was crashing
    - Enables application to fill these in later

4.0.0 PySimpleGUI & 2.0.0 PySimpleGUI27 19-June-2019

- DOC STRINGS DOCS STRINGS DOC STRINGS!
    - Your IDE is about to become very happy
    - All Elements have actual documentation in the call signature
    - The Readme and ReadTheDocs will be generated going forward using the CODE
    - HUGE Thanks for @nngogol for both copying & adding all those strings, but also for making an entire document creation system.
- New **version** string for PySimpleGUI.py
- New parameter to ALL `SetFocus` calls.
    - def SetFocus(self, force=False)
    - If force is True, then a call to `focus_force` is made instead of `focus_set`
- Get - New Radio Button Method. Returns True is the Radio Button is set
- Rename of Debugger class to _Debugger so IDEs don't get confused
- User read access to last Button Color set now available via property `Button.ButtonColor`
- Rename of a number of callback handlers to start with _
- Fix for memory leak in Read call. Every call to read lost a little memory due to root.protocol calls
- Listbox.Update - New parameter - scroll_to_index - scroll view so that index is shown at the top
- First PyPI release to use new documentation!

PySimpleGUI 4.1 Anniversary Release! 4-Aug-2019

NEVER has there been this long of a lag, sorry to all users! Long time coming. Docstrings continue to be a focus.

- Version can be found using PySimpleGUI.version
- New bit of licensing info at the top of the file
- Types used in the doc strings. Also type hints in some comments. Because also running on 2.7 can't use full typing
- Added using of Warnings. Just getting started using this mechanism. May be great, maybe not. We'll see with this change
- Added TOOLTIP_BACKGROUND_COLOR which can be changed (it's tkinter only setting however so undertand this!)
- Graph.DrawText. Ability to set `text_location` when drawing text onto a Graph Element. Determines what part of the text will be located at the point you provide when you draw the text. Choices are:
  - TEXT_LOCATION_TOP
  - TEXT_LOCATION_BOTTOM
  - TEXT_LOCATION_LEFT
  - TEXT_LOCATION_RIGHT
  - TEXT_LOCATION_TOP_LEFT
  - TEXT_LOCATION_TOP_RIGHT
  - TEXT_LOCATION_BOTTOM_LEFT
  - TEXT_LOCATION_BOTTOM_RIGT
  - TEXT_LOCATION_CENTER
- Flag ENABLE_TK_WINDOWS = False. If True, all windows will be made using only tk.Tk()
- SetFocus available for all elements now due to it being added to base class. May NOT work on all elements however
- Added Combo.GetSElectedItemsIndexes() - returns a list of all currently selected items
- Fixed Listbox.Update - set_to_index changed to be an int, list or tuple
- Added parent parameter to call to tkinter's askopenfilename, directory, filenames. Not sure why the root wasn't passed in before
- Button.Update - also sets the activebackground to the button's background color
- Graph - New parameter when creating. `float_values`. If True, then you're indicating that your coordinate system is float not int based
- Graph.Update - made background color optional parm so that visible only can be set
- Frame.Layout returns self now for chaining
- TabGroup.Layout returns self now for chaining
- Column.Layout returns self now for chaining
- Menu.Update menu_definition is now optional to allow for changing visibility only
- Added inivisiblity support for menu bars
- Table.Update supports setting alternating row color and row_colors (list of rows and the color to set)
- Set window.TimeoutKey to TIMEOUT_KEY initially

- Window - check for types for title (should be string) and layout (should be list) and warns user if not correct
- Window - renamed some methods by adding _ in front (like Show) as they are NOT user callable
- Another shortcut! Elem = Element = FindElement
- SaveToDisk - will not write buttons to file. Fixed problems due to buttons having keys
- Remapped Windowl.CloseNonBlockingForm, Window.CloseNonBlocking to be Window.CloseNonBlocking
- Fix for returning values from a combo list. Wasn't handling current value not in list of provided values
- Spin - Returns an actual value from list provided when Spin was created or updated
- Chaneged FillFormWithValues to use the new internal AllKeysDict dictionary
- Added try when creating combo. Problem happens when window is created twice. Prior window had already created the style
- Added list of table (tree) ids to the Table element
- Enabled autoclose to use fractions of a second
- Added a try around one of the destroys because it could fail if user aborted
- Popup - Icon is no longer set to default by default
- Fix for debugger trying to execute a REPL comand. The exec is only avilable in Python 3
- main() will display the version number in big letters when program is running

4.2 PySimpleGUI 2.2 for PySimpleGUI27 18 - Aug 2019

The cool lookup release! No more need for FindElement. You can continue to use FindElement. However, your code will look weird and ancient. ;-) (i.e. readable) MORE Docstring and main doc updates!

- Finally 2.7 gets an upgrade and with it doc strings. It however doesn't get a full-version bump like main PySimpleGUI as this may be its last release.
- New `window[key] == window.FindElement(key)`
- New Update calling method. Can directly call an Element and it will call its Update method
  `window[key](value=new_value) == window.FindElement(key).Update(value=new_value)`
- Made Tearoff part of element so anything can be a menu in theory
- Removed a bunch of `__del__` calls. Hoping it doesn't bite me in memory leaks
- Combo.Get method added
- Combo.GetSelectedItemsIndexes removed
- New Graph methods SendFigureToBack, BringFigureToFront
- Butten release changed for better Graph Dragging
  - Now returns key+"Up" for the event
  - Also returns the x,y coords in the values
- Tab.Select method added

- TabGroup.Get method added - returns key of currently selected Tab
- Window finalize parameter added - Will call finalize if a layout is also included. No more need for Finalize!!
- Quiet, steady change to PEP8 user interface started
    - Now available are Window methods - read, layout, finalize, find_element, element, close
    - Should provide 100% PEP with these alone for most PySimpleGUI programs
- Added finding focus across ALL elements by using the .Widget member variable
- Fixed sizing Columns! NOW they will finally be the size specified
- Fixed not using the initialdir paramter in PopupGetFile if the no_window option is set

4.3 PySimpleGUI Release 22-Aug-2019

PEP8 PEP8 PEP8 Layout controls! Can finally center stuff Some rather impactful changes this time Let's hope it doesn't all blow up in our faces!

- PEP8 interfaces added for Class methods & functions
    - Finally a PEP8 compliant interface for PySimpleGUI!!
    - The "old CamelCase" are still in place and will be for quite some time
    - Can mix and match at will if you want, but suggest picking one and sticking with it
    - All docs and demo programs will need to be changed
- Internally saving parent row frame for layout checks
- Warnings on all Update calls - checks if Window.Read or Window.Finalize has been called
- Warning if a layout is attempted to be used twice
  Shows an "Error Popup" to get the user's attention for sure
- Removed all element-specific SetFocus methods and made it available to ALL elements
- Listbox - no_scrollbar parameter added. If True then no scrollbar will be shown
- NEW finalize bool parameter added to Window. Removes need to "chain" .Finalize() call.
- NEW element_justification parameter for Column, Frame, Tab Elements and Window
    - Valid values are 'left', 'right', 'center'. Only first letter checked so can use 'l', 'c','r'
    - Default = 'left'
    - Result is that all Elements INSIDE of this container will be justified as specified
    - Works well with new Sizer Elements
- NEW justification parameter for Column elements.
    - Justifies Column AND the row it's on to this setting (left, right, center)
    - Enables individual rows to be justified in addition to the entire window
- NEW Sizer Element
    - Has width and height parameters. Can set one or both
    - Causes the element it is contained within to expand according to width and height of Sizer Element
    - Helps greatly with centering. Frames will shrink to fit the contents for example. Use Sizer to pad out to right size
- Added Window.visibility_changed to match the PySimpleGUIQt call

- Fixed Debugger so that popout window shows any newly added locals

4.4 PySimpleGUI Release 5-Sep-2019

- window() - "Calling" your Window object will perform a Read call
- InputText - move cursor to end following Update
- Shortcuts - trying to get a manageable and stable set of Normal, Short, Super-short
    - DD - DropDown (Combo)
    - LB, LBox - Listbox
    - R, Rad - Radio
    - ML, MLine - Multiline
    - BMenu - ButtonMenu
    - PBar, Prog - ProgressBar
    - Col - Column
- Listbox - new method GetIndexes returns currently selected items as a list of indexes
- Output - new method Get returns the contents of the output element
- Button - For Macs don't don't allow setting button color. Previously only warned
- ButtonMenu - new Click method will click the button just like a normal Button's Click method
- Column scrolling finally works correctly with mousewheel. Shift+Mouse Scroll will scroll horizontally
- Table - Get method is a dummy version a Get because Qt port got a real Get method
- Table - Will add numerical column headers if Column Headsing is set to None when creating Table Element
- Table - FIXED the columns crazily resizing themselves bug!!
- Table - Can resize individual columns now
- Tree - was not returning Keys but instead the string representation of the key
- SetIcon will set to default base64 icon if there's an error loading icon
- Fix for duplicate key error. Was attempting to add a "unique key counter" onto end of keys if duplicate, but needed to turn into string first
- Columns
    - No longer expand nor fill
    - Sizing works for both scrolled and normal
- Setting focus - fixed bug when have tabs, columns, frames that have elements that can get the focus. Setting focus on top-level window
- InputText elements will now cause rows to expand due to X direction expansion
- Frame - Trying to set the size but doesn't seem to be setting it correctly
- Tabs will now expand & fill now (I hope this is OK!!!)

4.5 PySimpleGUI Release 04-Nov-2019

- Metadata!
    - All elements have a NEW metadata parameter that you can set to anything and access with Element.metadata
    - Windows can have metadata too
- Window.finalize() - changed internally to do a fully window.read with timeout=1 so that it will complete all initializations correctly
- Removed typing import
- ButtonReboundCallback - Used with tkinter's Widget.bind method. Use this as a "target" for your bind and you'll get the event back via window.read()
- NEW Element methods that will work on a variety of elements:
    - set_size - sets width, height. Can set one or both
    - get_size - returns width, heigh of Element (underlying Widget), usually in PIXELS
    - hide_row - hides the entire row that an element occupies
    - unhide_row - makes visible the entire row that an element occupies
    - expand - causes element to expand to fill available space in X or Y or both directions
- InputText Element - Update got new parameters: text_color=None, background_color=None, move_cursor_to='end'
- RadioButton - fix in Update. Was causing problems with loading a window from disk
- Text Element - new border width parameter that is used when there's a relief set for the text element
- Output Element - special expand method like the one for all other elements
- Frame element - Can change the text for the frame using Update method
- Slider element - can change range. Previously had to change value to change the range
- Scrollable frame / column - change to how mousewheel scrolls. Was causing all things to scroll when scrolling a single column

    NOTE - may have a bad side effect for scrolling tables with a mouse wheel
- Fix for icon setting when creating window. Wasn't defaulting to correct icon
- Window.get_screen_size() returns the screen width and height. Does not have to be a window that's created already as this is a class method
- Window.GetScreenDimensions - will return size even if the window has been destroyed by using get_screen_size
- Now deleting window read timers every time done with them
- Combo no longer defaults to first entry
- New Material1 and Material2 look and feel color schemes
- change_look_and_feel has new "force" parameter. Set to True to force colors when using a Mac
- Fix in popup_get_files when 0 length of filename
- Fix in Window.SetIcon - properly sets icon using file with Linux now. Was always defaulting

Upcoming

Make suggestions people! Future release features

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.

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.

Named / Optional Parameters This is a language feature that is featured **heavily** in all of the API calls, both functions and classes. Elements are configured, in-place, by setting one or more optional parameters. For example, a Text element's color is chosen by setting the optional `text_color` parameter.

tkinter tkinter is the "official" GUI that Python supports. It runs on Windows, Linux, and Mac. It was chosen as the first target GUI framework due to its ***ubiquity***. Nearly all Python installations, with the exception of Ubuntu Linux, come pre-loaded with tkinter. It is the "simplest" of the GUI frameworks to get up an running (among Qt, WxPython, Kivy, etc).

From the start of the PSG project, tkinter was not meant to be the only underlying GUI framework for PySimpleGUI. It is merely a starting point. All journeys begin with one step forward and choosing tkinter was the first of many steps for PySimpleGUI. Now there are 4 ports up and running - tkinter, WxPython, Qt and Remi (web support)

Author & Owner

The PySimpleGUI Organization

This documentation as well as all PySimpleGUI code is Copyright 2018, 2019 by PySimpleGUI.org

Send correspondance to PySimpleGUI@PySimpleGUI.com

License

GNU Lesser General Public License (LGPL 3) +

Acknowledgments

There are a number of people that have been key contributors to this project both directly and indirectly. Paid professional help has been deployed a number of critical times in the project's history. This happens in the life of software development from time to time.

If you've helped, I sure hope that you feel like you've been properly thanked. That you have been recognized. If not, then say something.... drop an email to comments@PySimpleGUI.org.