

Learning PAGE - A Python GUI Designer



By G.D. Walters

SECOND EDITION

Now covers PAGE version 7

Version 7.0b 11/29/2021
Copyright © 2021 G.D. Walters

This document is licensed under the MIT License.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Table of Contents

Introduction.....	5
Some Background.....	6
Who this is written for.....	6
Why I wrote this guide.....	6
Some Conventions used in this document.....	8
Requirements.....	9
Installation.....	9
Windows.....	9
Linux and Raspberry Pi.....	9
OSX.....	10
Files Created by PAGE.....	10
Using an IDE.....	10
Starting PAGE.....	11
Windows.....	11
Linux, Mac and Raspberry Pi.....	11
What You Will Learn.....	12
Moving Forward.....	12
Chapter 1 - Getting Started.....	13
It's RAD!.....	14
Some Important Concepts.....	14
Designing The Form.....	16
Static Labels.....	17
Entry Widgets.....	18
The Button Widget.....	19
Looking at The Code.....	20
Editing Our Code.....	23
What You Have Learned.....	26
Chapter 2 - Moving on.....	28
Buttons.....	28
Frames.....	28
Labels.....	29
Single line entry text boxes.....	29
Check Buttons.....	29
Radio Buttons.....	29
Beginning the Project - Layout.....	29
Step 2 - Bindings.....	36
Step 3 – The Code.....	37
What you learned.....	42
Chapter 3 – A more realistic project.....	43
What you will learn.....	43

Building the UI.....	43
The Code.....	47
What you have learned.....	55
Chapter 4 – NASA Still Image Viewer.....	56
Pictures Pictures Everywhere.....	56
What you will learn.....	57
Getting Started.....	58
Building the GUI.....	58
The Code.....	64
Chapter 5 – Multiple Form Applications.....	74
What you will learn.....	76
Creating the GUI.....	76
Creating the Form Level Menu.....	76
Creating the Button Menu Bar.....	81
Adding our second form.....	87
Adding the third form.....	89
The Code.....	90
Conclusion.....	105

Introduction

A number of years ago I wrote two articles on how to use PAGE for Full Circle Magazine (issues 58 and 59 from back in February and March, 2012). At that time I was very involved in teaching Python to people who had very little background in programming, or intermediate knowledge of programming, but were new to the Python programming language.

Since then, I have been writing about using Python in the real world by controlling devices (sensors, motors, LEDs, etc.) on the Raspberry Pi and interfacing with the Arduino microcontroller. I recently completed my 121st “normal” Python article for Full Circle and the 7th article on using MicroPython for microcontrollers and am looking to continue for a good while.

When I wrote the original tutorial for PAGE, the released version was 4.11 way back in 2018. Now it is September 2021 and PAGE has changed quite a bit. At the moment, the release version is 6.2 and the original tutorial still works pretty well. However, Don Rozenberg has been very hard at work creating PAGE 7.0. Many things have change in PAGE between 6.2 and 7.0. While the interface that you use to create your GUI files, the entire code generator for PAGE has been rewritten and provides many new features.

I give many thanks to Don Rozenberg for his many years of creating and maintaining the PAGE program. I also want to thank my son Douglas for editing and sanity checking the original tutorial document and putting up with me as I go through writers block. Finally, allow me to thank a long time reader Halvard Tislavoll for keeping me on track with many things that I forget and take for granted.

Some Background

PAGE is a GUI designer for Python programmers using Tkinter that supports the Tk/ttk widget set written by Don Rozenberg. It is an extension of Visual Tcl that produces Python code.

Tcl stands for ‘**Tool Command Language**’ and is an open-source dynamic programming language. **Tk** is a graphical toolkit for **Tcl**. Tk can be used from many languages including C, Ruby, Perl, Python and Lua.

Tkinter stands for “**Tk Interface**” and is considered (according to the Python wiki) to be the de-facto standard GUI for Python. It is a thin object-oriented layer on top of Tcl/Tk. While Tkinter is not the only GUI programming toolkit for Python, it seems to be the most commonly used one.

While you can create Tkinter based GUIs without a designer like PAGE, it is a very tedious and, at least for me, a rather painful process. Programs like PAGE allow for rapid application development or **RAD**, making the process of development of graphical programs a fairly easy task. You can even include the end user, if one is available, to sit in on the design process of the user interface and make changes in real time before you get to the major coding portion.

Who this is written for

This guide is written for people who already know the basics of Python programming and want to expand their knowledge from terminal based applications to include Graphical User Interfaces or GUIs. If you have already used PAGE 6.2 or earlier, I’ve created a migration document that explains the changes between earlier versions of PAGE and PAGE 7. If you are one of these users, you might want to start there and then scan over this document just to catch something you might have missed.

This document is very similar to the tutorial I wrote for PAGE back in 2018. In fact, the projects for Chapters 2 and 3 are substantially the same projects, just updated for PAGE 7. I never really was happy with Chapter 1 in the original document. Once I released the tutorial, I felt that the project for Chapter 1 was very silly and really didn’t teach enough. When I started this tutorial, I decided to make it a bit more complicated to really show off the capabilities of creating a reasonable program using PAGE.

Why I wrote this guide

I’ve always enjoyed teaching, whether it is computers, cooking or coding. I feel like it’s my true calling in life. When I first got into computer programming in 1972, there wasn’t a great amount of information available for young people about programming, so I grabbed on to everything I could and I never stopped trying to learn more.

Python has become one of the fastest growing programming languages in the world and for the last few years has consistently been in the top 5 programming languages to know. With its popularity, there is a need for tools that allow for Rapid Application Development to produce user friendly interfaces beyond the command terminal. PAGE provides that in a free tool that is available to anyone with an Internet connection and the willingness to learn.

Some Conventions used in this document

For commands in a terminal window, I use **Liberation Mono 11pt Bold**.

Any code is shown in **Courier New Bold**. The font size will change from time to time.
Information that is considered important will be in **bold**.

When describing menu navigation, a '|' will be used to separate the various menu steps to traverse. An example of this would be **main menu | Gen_Python | Generate Support Module**.

Requirements

Back in PAGE 4.11, PAGE required a version of Tcl/Tk on your system in order to run. In PAGE version 6.0, this requirement was removed and PAGE is able to run using the Tkinter portion of Python without needing Tcl/Tk. This has made PAGE much easier to install and run on Windows, Linux and Mac. Many users were running Windows on a 32 bit machine and had major problems finding a version of Tcl that would work on a 32 bit Windows machine. Thankfully, this is no longer an issue and the only external requirement for PAGE to run is Python 3.6 or greater (Python's Tkinter must be at least 8.6). **PAGE no longer will run using Python 2.x.**

Installation

Windows

Thankfully, with the advent of PAGE 6.0, we no longer need to install Tcl/Tk, so that requirement is no longer needed.

Download the latest version of PAGE from <https://sourceforge.net/projects/PAGE/> and run the file. This will install PAGE. It will also modify your path so you can start PAGE from any folder on your system.

Finally, create a development directory on your hard drive to hold your files. I suggest somewhere on your root C: drive.

Linux and Raspberry Pi

Installation under Linux or the Raspberry Pi running the latest version of the Raspberry Pi Debian OS is much easier now than it ever was.

You will download the PAGE program from the source repository at <https://sourceforge.net/projects/page/>. It will be a .tgz file You will need to unpack

If you are using a Debian based Linux OS like Ubuntu, Linux Mint or even Raspberry Pi OS, you will have a `~/.bashrc` file. This file is loaded every time you start a terminal.

If you are using another Linux OS like Fedora, CentOS, you will want to check to find out what file you need to modify.

Modify your resource file to include an alias to PAGE. I always unpack PAGE to my Downloads folder into a folder called Page{version number}. Then I modify my `~/.bashrc` file to point to the proper folder like this...

```
alias page='python /home/greg/Downloads/Page-7/page/page.py'
```

This goes under the section that says `# some more ls aliases`.

After that, I save the file and issue a source ~/.bashrc in a terminal. This forces the ~/.bashrc file to be read for that terminal. Starting a new terminal will also include the alias to PAGE.

OSX

Again, from what Don tells me, the steps for installation are pretty much the same as Linux.

Files Created by PAGE

Once you have your GUI created and saved, PAGE will create three files. Let's pretend that our project is called ProjectX. The files created will be called ProjectX.tcl, ProjectX.py and ProjectX_support.py. Out of the three files you will ONLY want to modify the ProjectX_support.py file. The ProjectX_support.py file is where PAGE generates the callback function skeletons that you will edit and create any additional support functions.

Why? The ProjectX.tcl file is the file that PAGE creates that holds the GUI information for your project. If you ever need to update or change your GUI, this file will be rewritten by PAGE when you save the changes. The ProjectX.py file is called the GUI module. This file holds all of the basic information that is needed to display your GUI and is rewritten everytime you make any changes and regenerate the Python files. The ProjectX_support.py file provides all of the skeletons for your Python project. This includes all of the startup code to create the GUI program (like root = tk.Tk() and root.mainloop()) and things like the skeletons for any callback functions your project needs. (We'll see more about this in Chapter One.)

Using an IDE

PAGE creates the GUI buttons, forms and other widgets that make up your GUI. However, you have to write the "glue" that responds to what happens when a widget is clicked or modified. For this, you will need at least a text editor or Integrated Development Editor (IDE) to create and modify your code.

There are a number of free and paid for IDE programs out in the world. Python comes with an IDE called IDLE, which is written in Python and uses Tkinter for it's GUI. I've seen some issues between IDLE and PAGE programs, but you can use it to write your code.

There are others like Geany, VS Code, Atom and PyCharm as well as others that offer things like coloured syntax, code completion, and debugging that you can try and decide for yourself which one works best for you. Your school or employer might dictate your IDE, but for the most part the IDE that you end up using, is your choice.

I'm going to leave the installation of your IDE/editor to you.

One last thing about IDE programs. They are not created equal. While there are dozens out there, some have hidden “options” that might make your job harder rather than easier. While this is strictly a personal opinion, I would suggest not using IDLE as your editor of choice. There is a “feature” that starts Python with the `-i` to enter the interactive mode after the program is finished. This leaves the UI on the screen and might cause some concern (Thanks Halvard for pointing this out to me).

Starting PAGE

Starting PAGE correctly is very important. This varies from machine Operating System to Operating system, so I’ll describe the process here for each OS.

The very first thing you need to do is to create a development folder that you will keep all your projects in. It is VERY important that whenever you work with PAGE, that you work out of a development folder that contains all of your resource files (like images, sound files, database files, etc.) in the folder for a specific project.

Windows

When you installed PAGE, the installation program modified your path statement to include the proper path to PAGE. It also created an icon on the Desktop that will start PAGE. However, I suggest that you don’t use the desktop icon to start PAGE, since that will use the desktop as the path for the project.

The best way to start PAGE under Windows is to navigate to your project folder in your development folder. You can do this one of two ways. Either open a command prompt, navigate to your project folder and then start PAGE or use the Windows File Explorer to go to your project folder then press {Ctrl}+ L to open the address bar near the top of the Explorer window. Once there, simply type CMD which will open a command prompt for you. Then simply type page.

You can include the filename of the tcl file (“**page myproject.tcl**” or simply “**page myproject**”) or other options. By doing things this way, all internal references to resources are from that spot. This will allow you to move all of your program code and resources to another folder after you are done.

Linux, Mac and Raspberry Pi

Like with Windows, you need to create a project folder that will include all your resource files (images, sound files, databases, etc.) that your program will need to run properly.

Either use a terminal to navigate to your project folder or Nemo or your file manager to get to your project folder and use the <Open In Terminal> option.

What You Will Learn

In Chapter 1, I'll take you through the steps to create a simple login GUI and write the code that allows you to create an entire, but fairly trivial program that runs.

In Chapter 2, you will learn how to create another simple GUI that uses many of the widgets that are used in every day programs, how to set up the attributes for each of those widgets, and how to modify some of those attributes in code.

In Chapter 3, you will learn to capitalize on the skills that were learned in the first two chapters to create a real world program to search for media files.

In Chapter 4, we will talk about using a Label widget to display images that are pulled from the NASA web site.

Finally, we wrap up in Chapter 5 learning about creating menus for your applications and creating applications that have multiple forms.

The source code for each chapter is included in a zip file in the /page/docs/tutorial/Learning Page/ folder within your distribution.

Moving Forward

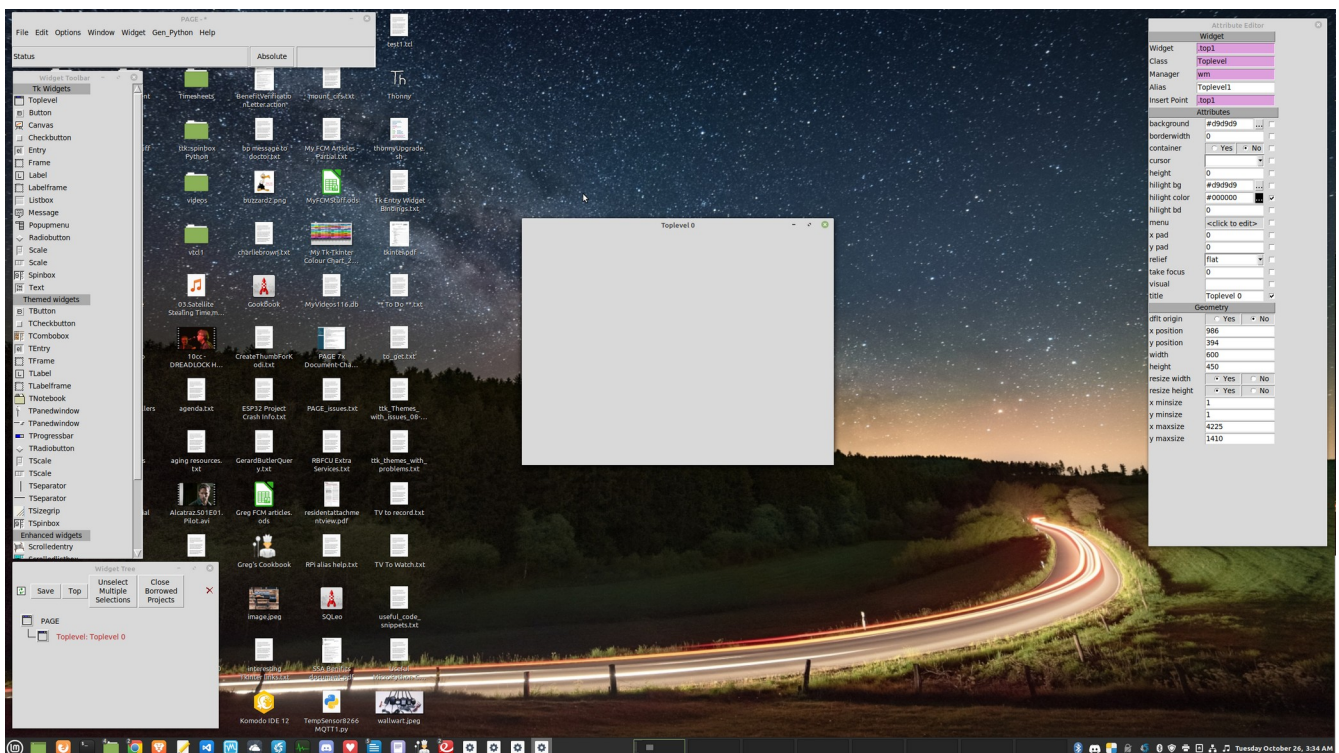
Now that PAGE is loaded on your machine and you know how to start PAGE, let's move to Chapter 1 to create our first program.

Chapter 1 - Getting Started

We've already talked about the correct way to open PAGE. So, let's start off by making a work folder. Somewhere on your computer, create a folder called PAGETutor (all one word). Open that folder and create another one named Chapter_1.

Please don't use spaces in either your folder names or your filenames for anything you do with PAGE. There are times that these will become Python identifiers and they can not contain spaces.

Now change to that folder and open a terminal window or command prompt, and start PAGE. You should see



something that looks like this...

Please excuse my messy desktop, but it's the way I work best. I've minimized the terminal window and the File Manager windows to focus on the PAGE application. We'll take a quick look at each of the five windows that makes up PAGE and what they do.

Starting at the top left and moving clockwise, the windows are:

- The Main PAGE window (top left)

- The Attribute Editor (right edge)
- The Widget Tree (bottom left)
- The Widget Toolbar (between the Main window and the Widget Tree)
- Finally in the middle is the design Toplevel form.

The Main window allows you to save your project, open existing projects, generate the Python code and a whole lot more.

The Attribute Editor changes it's appearance or behavior depending on what widget you are working on. Some widgets only have a few options while others have so many, the window needs to show a scroll bar on the right edge so you can get to them all.

The Widget Tree shows every widget that you've put into your form and it's relationship to the ones around it. We'll look at this in more detail after a bit.

The Widget Toolbar is where you can find all of the widgets (or controls) that you can use to build your GUI. I like to call it "the toolbox". I'll use that term often.

And last but not least the design form. It's called the Toplevel form and it is actually a widget in itself. You'll place all of the widgets that make up your GUI on this.

It's RAD!

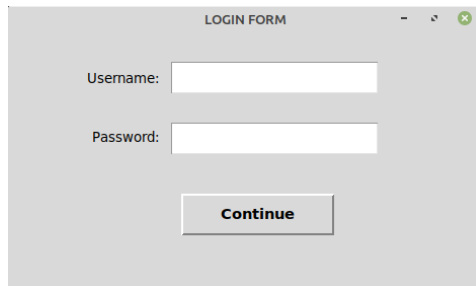
RAD stands for Rapid Application Development (or Designer) and that is exactly what PAGE is and does. It allows you to create the entire GUI, save it and the PAGE will generate all the Python code for you.

Now before you get all excited, I need to clarify something. When I say PAGE will generate all the Python code for you, that's true but only to a certain extent. Once you have generated the code, the Python code will run and you can see the GUI of your program. Buttons will respond to a mouse click, spin boxes will move. However, your program really won't do anything. PAGE creates the Visual portion but it's up to you to "write the code that glues all the widgets together".

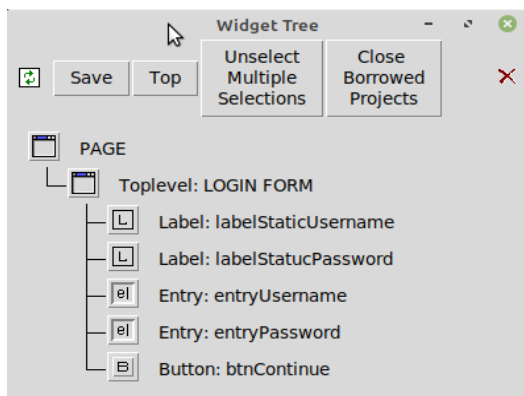
Some Important Concepts

Don't start changing anything on your form yet. This section is just discussion. We'll change things in the next section.

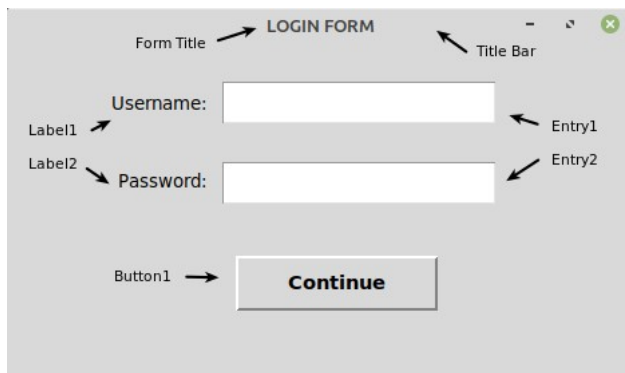
Every widget, including the Designer form which is called the Toplevel Widget, gets a name generated by PAGE when it's added to the form. These names while somewhat descriptive, are like **Button1** or **Entry4**. This generic name is called the Alias and is the way you would reference the widget when you want to or need to change some of it's attributes or get information from the widget when your code is running. While they tell you what the widget is, it doesn't tell you what the widget is for. A good example would be something like this...



It's a simple login form that asks the user for the user's name and the user's password and then has a button that allows the user to continue. Now let's take a look at the widget tree to see how the various widgets are named.



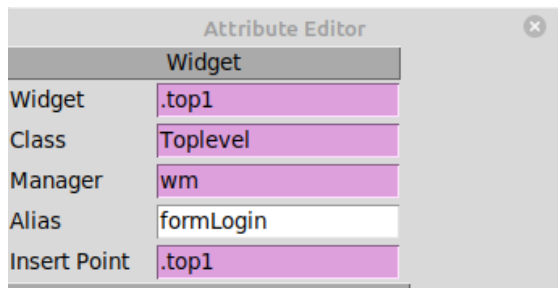
The widget tree does show the widget types and the names that PAGE created as they were placed on the designer Toplevel form. But what this doesn't tell us is what each widget is used for. I've taken the form capture and annotated it.



In a form this simple, it would be easy to remember that **Entry1** is for the username, **Entry2** is for the password and **Button1** is to continue on to the next form or process. But if someone else were to look at your code, they would have no idea what those widgets were for.

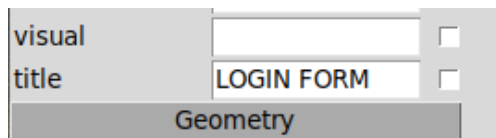
Designing The Form

It's easy to fix this, though. Let's rewind a little bit and start with our blank Toplevel form. Look at the Attribute Editor near the top.

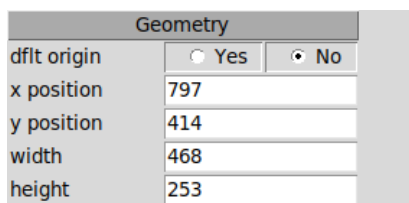


Notice the Alias box (It's actually an **Entry Widget** by the way). Since we are working on the Toplevel form, we can simply type the name of our form. Just like "normal" widgets, it's important to name our form as well. Also notice that all the other four boxes are marked with a Plum or purplish colour. This is because those boxes are not supposed to be changed under normal circumstances, and this is a normal circumstance.

Next, look down the the other entry boxes in the Attribute Editor for our Toplevel. You'll see the title section.



Change that to **LOGIN FORM**. Now we want to set the size of our form. You can resize the Toplevel form by moving to one of the edges or corners and doing a click and drag to set the size you wish, or manually set the size in the Geometry section of the Attribute editor. For now, we'll manually set the size.



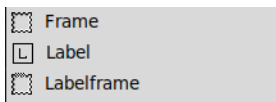
Set the width to 468 and the height to 253 as shown in the above picture. That's all we need to change for our Toplevel form. Before we go any further, save your project. You can do this by going up the the main window and selecting **File | Save** and then entering the name of the file in the dialog box. For this project, we'll call it **login.tcl**. You can leave off the **.tcl** extension if you wish, but I like to be sure that I include it. Remember to save early and save often.

Tip: Don implemented an autosave feature in PAGE 6.1 and has continued to improve it in PAGE 7. While the autosave feature works well, do not rely on autosave. It's always good to remember to save often!

Now we can think about putting some other widgets in. However, before we actually do, it's important to know that when you have a project form that takes a lot of text entry from the user of your program, like this one, that PAGE remembers the order that you put the widgets into the form. Many widgets can take focus (which allows the user to enter data or change things). This is called **Tab Order**. When you or the user of your program enters data, and they have finished that the entry for that widget, they can simply press the {**Tab**} key to move to the next widget to continue the data entry. Not every widget can take focus. Standard static text widgets don't normally get to take focus, since the text in the label stays the same during the life of the form. But widgets like the Entry widgets and even Buttons can take focus. So be sure to build this form the same way I lay it out here.

Static Labels

As you can tell from the images of our completed form above, there are two Static Labels on the form. These are used to give the user an idea what the widget(s) around it are for. Click on the Label icon in the toolbox. This tells PAGE that you are planning on putting a Label widget on the form.



Now click somewhere in the container section of the Toplevel to place the widget. From here you can simply drag the widget close to where you want it to “live”. Notice the Attribute Editor has automatically changed quite a bit. All of the options in the Attribute Editor now are for the Label Widget. Now follow the grid below to set the attributes for our first Label Widget.

Attribute	Value
Alias	labelStaticUsername
anchor	e
text	Username:
x position	56
y position	37
width	99
height	21

For this project, these are the only attributes you need to set. Once you get through you should notice all the settings have already been applied. The anchor attribute set the position of the text within the widget itself. The **e** stands for “**E**ast” which means that the text should end at the right side of the widget with blank space on the left side. This is the edge of the widget that is close to the Entry widget that we will be placing soon. By default, the anchor attribute is set to “**W**est” (w). There are other anchors you can set, but you can experiment with the anchor setting on your own.

Now do the same thing again for our second Label. The attributes for the second Label will be a bit different.

Attribute	Value
Alias	labelStaticPassword
anchor	e
text	Password:
x position	56
y position	95
width	99
height	21

You might notice that the Label widgets we just put on our form have the same width and height and also the same x position. This makes them line up correctly visually on the form.

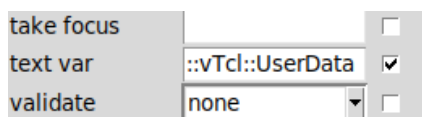
Entry Widgets

Next we will put the two Entry widgets on the form. Just like with the Label widgets, simply click on the Entry widget in the tool box and click the container section of the form somewhere close to where you want the widget to end up. I'll provide another set of grids for you to know what the attributes should be.

Attribute	Value
Alias	entryUsername
cursor	xterm
text var	UserData
x position	160
y position	30
width	206
height	33

You have already noticed that some of the attributes are different for the Entry widget than what we had for the Label widget. Remember, the Attribute Editor always changes and shows you the proper attributes that reflect the widget you currently have selected.

Your Attribute Editor might have changed the information in the text var attribute to something that looks like this...



It's ok, so don't worry about it. It's just an artifact that goes along with the Visual Tcl program the Don uses to create PAGE. He's working hard to change it. The bottom line here, is that your actual program will only contain **UserData**.

Now, add another Entry widget and set the attributes as shown below.

Attribute	Value
Alias	entryPassword
cursor	xterm
text var	PassData
x position	160
y position	90
width	206
height	33

By now, you should have figured out that the value for x position is the right/left position of the widget and the y position is the up/down position. If you haven't already, save your work.

The Button Widget

The last thing we need to do is to place a Button widget on the form. Make sure it is the Tk Button, not the Themed Widget TButton. Here are the attributes you need to set for the Button widget.

Attribute	Value
Alias	btnContinue
anchor	center
command	on_btnContinue
text	Continue
x position	170
y position	160
width	153
height	43

We've set the attribute for a new item, called command. This will create a function in our code module called **on_btnContinue**. This function is called a **Callback function**, which gets called in our program whenever the button is clicked by the user. PAGE will create a skeleton of this function for us, so that when we get to writing our code, it's already started for us.

We will be adding a few attribute settings later on, but for now save your project again. Now we will generate our Python code. To do this, go back to the Main Window, click on **Gen_Python | Generate Python GUI** menu item. After a couple of seconds, a new window will pop up with our base Python code in a simple editor. Feel free to scroll through the code listing, but **DO NOT** change anything. Click the Save button to save the file.

Back in the Main Page window, click again on **Gen_Python | Generate Support Module**. This will open the Generated Python window again, this time with the rest of the code that you will need to run our application. Again, feel free to scroll through this file. These editors are only designed to display your code to you and you

can make simple changes if needed, but you really shouldn't use them like a real editor. Again, click Save to save the code file.

Now you can close PAGE and you should notice that our project folder should contain three files, login.tcl, login.py and login_support.py. The login.tcl file is the actual Tcl GUI definition that PAGE uses. This should NEVER be edited by hand. The login.py file is the GUI definition file for Python and again, should NEVER be edited by hand. The last file, login_support.py is the file that we will be editing.

Looking at The Code

Open the login_support.py file in a text editor or your IDE. It should look pretty much like what I show below. I'll break in with some comments and explanations.

First, we have the import section. For our current project nothing needs to be added. PAGE has put it all in for us.

```
import sys
import tkinter as tk
import tkinter.ttk as ttk
from tkinter.constants import *

import login
```

Please notice the `from tkinter.constants import *` line. This is new to PAGE version 7. It's an important addition. In previous versions of PAGE, when you wanted to use a constant in your code, you would have to use something like `Tk.END` or `TK.CENTER` if you wanted to refer to the constants. Now you can simply use `END` or `CENTER` in your code. The constants file is located in your Python folder usually in the `tkinter` folder. For example, if you use Linux Mint and your Python version is 3.9, the path would be `/usr/lib/python3.9/tkinter/constants.py`. You can also do a web search to find all of the Tkinter constants.

The next batch of code after the import section is the main function. This is the entry point for your program and is run just before the GUI is shown to the user. Notice that there are two lines of comments in this section of code. The obvious one to people who use Python often starts with a `"#"`. The not so obvious one is this one

```
'''Main entry point for the application.'''
```

Commenting your code is VERY important. Not only does it let other people know what you are thinking as you write the code, but if you come back six months or a year later you will be able to remember what you were thinking.

```
def main(*args):
```

```
'''Main entry point for the application.'''
global root
root = tk.Tk()
root.protocol( 'WM_DELETE_WINDOW' , root.destroy)
# Creates a toplevel widget.
global _top1, _w1
_top1 = root
_w1 = login.formLogin(_top1)
root.mainloop()
```

Notice the last four lines of the main function. The first one sets two global variables, **_top1** and **_w1**. In versions of PAGE before version 7, these two variables used to be called **top** and **w**. The reason they were changed is to support multiple forms within the support Python module. You really don't need to be concerned about this at this point, we'll cover it in Chapter 4.

The third line of the four sets **_w1** to the form we are working with. Any time we want to reference a widget on the form or a variable we've set up, we have to preface it with "**_w1**." You'll see this in a moment.

If you want to run any code before the program is shown to the user, you need to add a call to your start up code between the last two lines of the **main** function. The last line (**root.mainloop()**) is the actual line that shows the form and starts the Tkinter loop. We'll do this in just a little bit.

This next function is the callback that we asked PAGE to create for us. This is the function that will run whenever the user clicks the continue button.

```
def on_btnContinue(*args):
    print('login_support.on_btnContinue')
    for arg in args:
        print ('another arg:', arg)
    sys.stdout.flush()
```

Notice that it's just a skeleton, so we'll have to add to this function to make it do much more than to print that it's been called. We'll do that in a minute. The code that prints the function name and the arguments passed to the function in the terminal can be commented out or deleted entirely, however, it provides a visual indication that your callbacks are working and it can be very valuable when debugging your code.

First, we need to look at the very last bit of code. This is where the program actually starts. This is here so that no matter if you start the program by calling `login.py` or `login_support.py`, it will start correctly. In reality, you should start the program by calling the `login.py` file.

```
if __name__ == '__main__':  
    login.start_up()
```

Now that we've seen the support file, we should take a short look at a little bit of the login.py file, just so you have an idea what is happening in there. This file also contains an import section. I've put this part of the code in "normal" typeface since you really don't need to pay too much attention to it. It's just here for reference.

```
import sys  
import tkinter as tk  
import tkinter.ttk as ttk  
from tkinter.constants import *  
import login_support
```

Next is the actual class definition for our GUI program. This holds the code to create each of our widgets (including the Toplevel widget), where they are supposed to be on the Toplevel form and any attributes that we set up during the design time process using PAGE.

```
class formLogin:  
    def __init__(self, top=None):  
        '''This class configures and populates the toplevel window.  
        top is the toplevel containing window.'''  
        _bgcolor = '#d9d9d9' # X11 color: 'gray85'  
        _fgcolor = '#000000' # X11 color: 'black'  
        _compcolor = '#d9d9d9' # X11 color: 'gray85'  
        _ana1color = '#d9d9d9' # X11 color: 'gray85'  
        _ana2color = '#ecec' # Closest X11 color: 'gray92'
```

The next line (top.geometry...) provides the width, height and the x and y positions of the form when you saved it.

```
        top.geometry("468x253+797+414")  
        top.minsize(1, 1)  
        top.maxsize(4225, 1410)  
        top.resizable(0, 0)  
        top.title("LOGIN FORM")
```

```
top.configure(highlightcolor="black")
```

This next section is very important if you have created any text variables (text var) or any variables. Remember we set the text vars for the two Entry widgets. PAGE puts that information here, so we can use in our code.

```
self.top = top
self.UserData = tk.StringVar()
self.PassData = tk.StringVar()
```

From here down, are the definitions of each widget, it's size, placement, and any other configurations. I'm only including one here, just so you can see.

```
self.labelStaticUsername = tk.Label(self.top)
self.labelStaticUsername.place(x=56, y=37, height=21, width=99)
self.labelStaticUsername.configure(activebackground="#f9f9f9")
self.labelStaticUsername.configure(anchor='e')
self.labelStaticUsername.configure(text=' 'Username:' ')
...
```

I'm not going to bore you with all the code for the GUI.py module. You can look at it yourself if you want to.

Editing Our Code

Now, let's go back to our support file in our editor. We need to make a few small changes before we try to run our program.

The first thing we are going to do is to add a line to the import section. You want to add the following line.

```
import tkinter.messagebox as messagebox
```

This gives us the ability to show pop-up message dialog boxes to let the user know something like warnings, errors or just information.

Next, we are going to modify the callback function to actually do something. Here is the code. The part in bold text is the part you are going to add. I'll explain as we go.

```
def on_btnContinue(*args):
    print('login_support.on_btnContinue')
```

```
for arg in args:
    print('another arg:', arg)
sys.stdout.flush()

users = ['Greg', 'Steve', 'Sam']
password = 'password'
```

In the first two lines, we create a list of “approved users” and a password. Feel free to change the list of usernames and the password if you want to. I suggest you do this after we finish this chapter of the tutorial.

We need to check the text var **UserData** by using the **.get()** method to retrieve the data. However, in order to do so, we need to preface it with **_w1.** which is the way we get to the text var. The same goes for the **PassData** variable. So first, we compare the information that the user has entered with the data in the users list. Then we check to see if the password that was entered to our “default” password, **password**. If both match, we set up a title (**titl**) and message (**msg**) to fill in a message box and use the **message.showinfo()** function to show the **messagebox** to the user. When they click Ok, we end the program politely using **root.destroy()**.

```
if _w1.UserData.get() in users:

    if password == _w1.PassData.get():

        titl = 'Login Accepted'

        msg = 'Username and password accepted'

        resp = messagebox.showinfo(titl, msg)

        root.destroy()
```

If one or the other doesn’t match, we set up the information to show an error messagebox, then we use the **.set(“”)** method to clear both of the Entry widgets and use the **.focus_set()** method of the Entry widget to give the user another chance.

```
else:

    titl = 'Login Information Incorrect'

    msg = 'Information not accepted'

    resp = messagebox.showerror(titl, msg)

    _w1.UserData.set('')

    _w1.PassData.set('')

    _w1.entryUsername.focus_set()
```

The final things we need to do is modify the **main** function to include a short start up function. There are only two lines (if you count the comment) that need to be added here. They are shown in bold below.

```
def main(*args):

    '''Main entry point for the application.'''
```

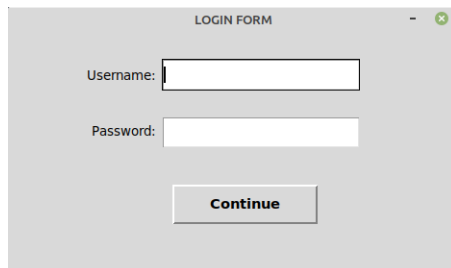


```
global root
root = tk.Tk()
root.protocol('WM_DELETE_WINDOW', root.destroy)
# Creates a toplevel widget.
global _top1, _w1
_top1 = root
_w1 = login.formLogin(_top1)
# My startup code here
start_up()
root.mainloop()
```

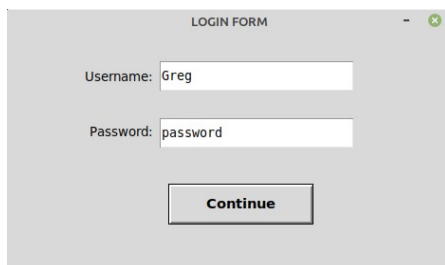
The start_up function will set focus to the username Entry widget so that when the program starts, the user can just start typing.

```
def start_up():
    _w1.entryUsername.focus_set()
```

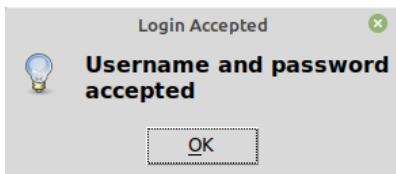
That's it. Now, save your Python file and run your program. Your form should look like this.



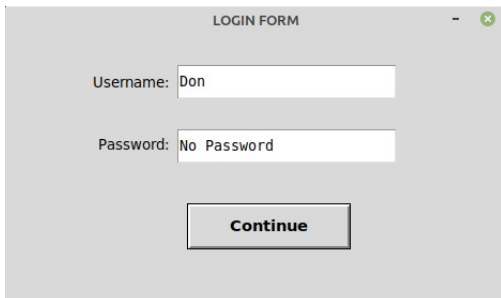
Notice that the cursor is already in the Username Entry widget. Type "Greg". Press the {Tab} key. The cursor moves to the password Entry widget. (Remember the Tab order I talked about.) Type "password". Press {Tab} again. The Button is now selected.



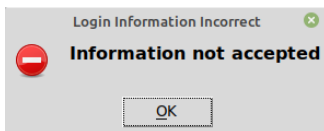
Press the {Spacebar}. Message box should appear.



Click the OK button. Program should end.



Restart the program and type a username that is not in the user list. Click in the password Entry widget or press {Tab}. Enter a "bad" password. Press {Tab}. Click the Continue Button (or press {Spacebar}). Error messagebox should appear.



Click OK on the Error box. Entry widgets should clear and Username Entry should get focus.

You've created your first application in PAGE. CONGRATULATIONS!

What You Have Learned

- How to properly start PAGE.
 - Always remember, start PAGE from the development folder for your project.

- Keep every project in a separate folder.
- Selecting and placing widgets on the Toplevel form.
- Using the special Tk variables.
 - Widgets like Checkbuttons and Radiobuttons use the **tk.IntVar**. We'll see Checkbuttons and Radiobuttons in the next chapter, but it's a good thing to know.
 - Widgets like the Entry widgets use a **tk.StringVar**.
 - Use **_w1.Variable.get()** to retrieve the value.
 - Use **_w1.Variable.set()** to set the value.
- If you need to set things up before the user sees the GUI, call your set up routine from the next to last line in the **main** function.

In Chapter 2, we'll look at using other widgets like Checkbuttons, Radiobuttons, Dynamic Labels and so on.

Chapter 2 - Moving on

Our next project will create a window with the following widgets...

- Buttons
- Frames
- Labels
- Text Box (Single line entry widget)
- Check Button
- Radio Button

This simple program will perform the following tasks..:

- Frames: Demonstrates the grouping widgets into logical visual sets.
- Entry widget: Demonstrates a way for the user to provide data that can be retrieved programmatically, in this case a button. Clicking on the button gets the information from the entry widget and prints it in the terminal window.
- Radiobuttons: Demonstrates the grouping function of the radio buttons and setting a label with text dynamically. This also demonstrates the “one of many” nature of Radiobuttons.
- Checkbuttons: Demonstrates the On/Off function of check buttons and will print in the terminal window the value of the checked or on widgets upon clicking the associated button.
- Label widget: Demonstrates the ability of dynamically changing the text displayed to the user.

Before we get started, we should explore what each of these widgets do.

Buttons

Buttons, as we discovered in our first program example, are widgets that when clicked (pressed either with a mouse button or a keyboard keypress or even tapped with a finger on a touch screen) raises an event that our program will act upon by jumping to a callback function. We all are familiar with buttons.

Frames

Frames provide a simple way to group items that logically belong together. A Labelframe is simply a frame with a label attached that provides immediate indication as to the intent of that grouping.

Labels

Labels are usually static text that shows the end user what a particular widget is for. It is static since it never changes (or rarely changes) during the life of the program. There are times that you might want to use a label as a dynamic display to show what is happening somewhere in the program (which is what we will do with the project in this chapter).

Single line entry text boxes

An entry box is a widget that allows the end user to type in information or data. There are two different types of text entry widgets in the standard Tkinter widget library, an entry widget and a text widget. Use the entry widget for single line data entry like first name, last name, address, etc. When you need multi-line data entries, like a series of instructions, use the text widget.

Check Buttons

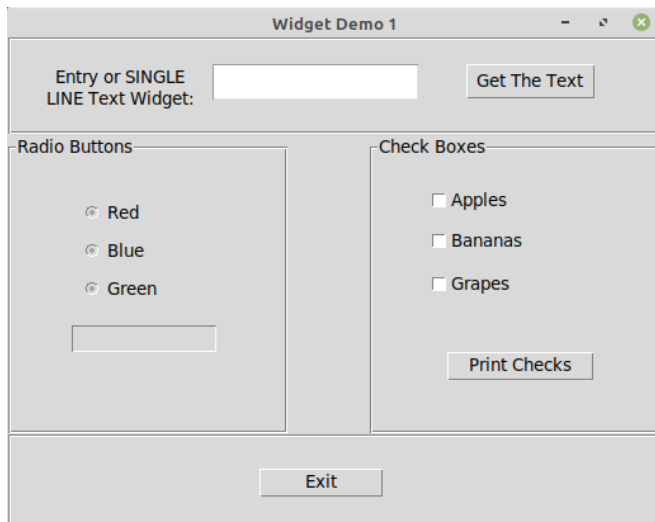
Check boxes provide a visual reference to a Yes/No or On/Off status of a variable or option. Check boxes are standalone and don't normally interact with any other widgets. Consider them as a Many-Of-Many visual tool. Many of the check boxes can be selected, or checked, at one time.

Radio Buttons

Radio buttons, like a Check box, provides a visual reference to a Yes/No or On/Off status, but are usually part of a One-Of-Many widget set. Only one Radio button may be selected (On) state within a group at any time and are usually grouped together within a frame. When one radio button is selected, all others in the group are de-selected. All radio buttons in a group use the same variable name. The value attribute will be used to specify what each button will send to that variable.

Beginning the Project - Layout

It's always good to have an idea of what your UI will look like before you get started. I like to sketch my design on paper first, just to get an idea of what I'll need in the way of space and sizes. Here is what our finished project will look like in our PAGE designer...



As you can see, we need four frames, two of which are Labelframes and two that are regular frames. We will also need three standard Buttons, three Radiobuttons, three Checkbuttons, one single line entry widget and two labels. You might not immediately recognize the second label widget. It is the sunken rectangle on the left. It has no label right now, since the text will be dynamically generated by the selection of the radio buttons. We will set up our GUI first with all the widgets and their attributes, then we'll set our bindings and finally write the code required.

Start PAGE and move the New Toplevel widget so it is centered in your screen and save your project right away. Call it "SecondApp".

I'll give you a list of attributes for each of the widgets we use, that need to be set in the Attribute Editor. The value you should use is in bold. The X position, Y position, Width and Height are set in the Geometry section of the Editor. Here is the value set for our Toplevel window widget.

Attribute	Value
Alias	Toplevel
title	Widget Demo 1
x position	344
y position	162
width	517
height	386

Next, we need our first frame. Click on the frame button in the toolbox and place it anywhere in the Toplevel widget. You will set its location, alias and size in the attribute editor with the following information:

Attribute	Value
Alias	frameTop
x position	1

y position	0
width	515
height	76

Tip: In addition to entering all of the attributes in the Attribute Editor, you can right click on the widget and quickly set **some** of the more important attributes from a pop-up context menu. This includes the Alias as well as other widget specific attributes like text for a button widget. You can do the same thing from the Widget Tree window.

Now we will populate the top frame with a standard Label widget, an Entry widget and a standard Button widget. Start with the Label widget and place it somewhere into the left side of the top frame. Set its attributes to the following values:

Attribute	Value
Alias	Label1
title	Entry or SINGLE LINE Text Widget:
wrap length	130
x position	10
y position	10
width	156
height	61

Note: When setting the Wrap length of a widget (for those widgets that support it), you will be using a value of screen units (pixels). You might have to tweak this value to get it to look exactly the way you want.

Next place an Entry widget near the middle of the frame and set the attributes as follows:

Attribute	Value
Alias	entryExample
x position	160
y position	20
width	164
height	30

Finally, place a standard button near the right side of the frame. The attributes should be:

Attribute	Value
Alias	btnGetText
text	Get The Text
x position	360

y position	20
width	103
height	29

Note: The width and height are dynamic in this case and already set for us when we entered the text for the button. You can always manually set the values to suit your needs.

Now we will deal with the left side of the window that will hold the Radiobuttons. We'll start by placing a Labelframe widget near the middle of the Toplevel window. Once that is done, set its attributes as follows:

Attribute	Value
Alias	lframeRadioButtons
text	Radiobuttons
x position	1
y position	77
width	220
height	235

Tip: If you accidentally set a value in the relative x or relative y entry boxes, things will probably not look the way you expected (like the widget disappearing). If this happens, simply put a "0" (zero) in the entry box you changed and things will go back to normal.

Next we need to place three Radiobutton widgets into the left frame. Feel free to put them all in at one time and then go back and set the attributes. If you do that, you can use the widget tree to select which button to work with rather than clicking each widget in the designer. Be sure you set all the attributes that I show.

Attribute	Value
Alias	rbRed
command	on_rbClick
text	Red
value	1
variable	Colors
x position	50
y position	50
width	56
height	23

NOTE: The Radiobutton value variable can be either a string or an integer. Many times, an integer works fine. PAGE will determine what type of variable based on what you enter in the value attribute. You **MUST** make sure that all the values are the same type or your program won't work properly.

Now for the second Radiobutton...

Attribute	Value
Alias	rbBlue
command	on_rbClick
text	Blue
value	2
x position	50
y position	80
width	60
height	23

And the third Radiobutton...

Attribute	Value
Alias	rbGreen
command	on_rbClick
text	Green
value	3
x position	50
y position	110
width	70
height	23

The last thing we need to put into this Labelframe is the label that shows the status value of the selected radio button. As usual, place it somewhere in the left frame that is empty. Also notice that I left the text of the label empty or blank. That's because we'll use the text var attribute to change the label text from our program code.

Attribute	Value
Alias	lblRbInfo
text	
text var	ColorInfo
x position	50
y position	150
width	114
height	21

At this point, you should save your project. As I said in Chapter 1, save often.

Next, we need another Labelframe on the right side of our UI. This will hold our Checkbuttons and a standard button.

Attribute	Value
Alias	IframeCheckBoxes
text	Checkbuttons
x position	286
y position	77
width	230
height	235

Now add three Checkbuttons and set the attributes as follows. For the first Checkbutton ...

Attribute	Value
Alias	chkApples
anchor	w
text	Apples
text var	
variable	che39
x position	40
y position	40
width	83
height	23

NOTE: PAGE will automatically populate the variable attribute for you, if you add the Checkbutton from the toolbox. If you copy and paste a Checkbutton widget, the variable attribute will be the same as the original widget.

For Checkbutton #2

Attribute	Value
Alias	chkBananas
anchor	w
text	Bananas
text var	
variable	che40
x position	40
y position	72
width	83
height	23

Checkbutton #3

Attribute	Value
Alias	chkGrapes
anchor	w
text	Grapes
text var	
variable	che41
x position	40
y position	106
width	83
height	23

Finally add the standard button. When this button is clicked it will call some code that will print which buttons, if any, are checked.

Attribute	Value
Alias	btnPrintChecks
text	Print Checks
x position	60
y position	170
width	117
height	24

You might notice that this button does not have the command attribute filled in to create our callback function like we did in Chapter 1. There are two ways to link callback functions for Buttons and some other widgets. One is the command attribute which we showed when we did the Radiobuttons. This would also work for the Checkbuttons, but we didn't use it, since we will use the button that we just set up. The other way is to use the Bind command. There is a nice way that PAGE provides to work with the bind function, which we will explore in a few minutes.

Now, save your program again, just in case.

The last thing we need to do in our layout process is to put another standard frame at the bottom of our form. This will hold a standard button to close the application.

Attribute	Value
Alias	frameButton
x position	1
y position	313
width	515
height	72

Last but not least, place the standard button near the middle of the bottom frame.

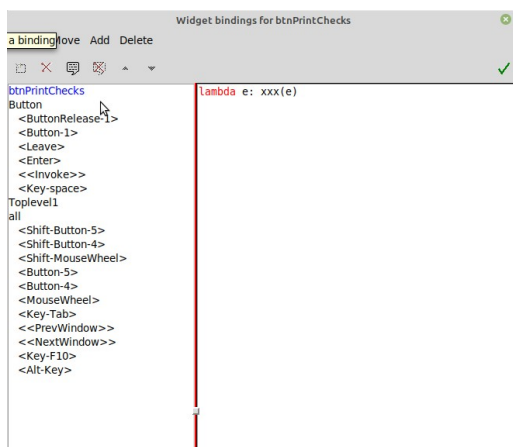
Attribute	Value
Alias	btnExit
text	Exit
x position	197
y position	26
width	99
height	24

Now be sure to save your project and generate the Python code and, if you wish for now, the support module. That being done, we can move on to the bindings of our widgets to the code routines.

Step 2 - Bindings

Luckily, most of our bindings are already done for us, thanks to PAGE. But there are a few things that still need some attention.

At this point, the only bindings that we need to deal with will be our standard buttons. We'll start with the top most button, **btnGetText**. Right click (Mouse button 2) on it, being careful that you don't move the button within the frame, and select '**Bindings**' from the context menu that pops up. (Alternately, you can right click on its entry in the Widget Tree window. Personally, I like to use the Widget Tree method, so I make sure I don't accidentally nudge the widget. I hate it when I spend a lot of time making the GUI look just right then goof it up by having shaky hands trying to bind an event to a widget. You can't undo the change in position of a widget.)



You will see our selected widget listed at the top of the left panel in blue. We want to bind a Button-1 (left click) event to it. Make sure you click the blue widget alias in the left panel. There are two ways to add an event. You can either click Insert from the menu and then select the event you want from the drop down menu or you can

click the small icon on the far left from the button bar which brings up the same menu. At this point, you want to select the item that says “Button-1”.

Tip: Most modern mice support three buttons and sometimes more, but the numbering can be confusing. If you look at the mouse, the button on the left is Button-1. The button on the right is Button-2. If your mouse has a scroll wheel, it usually is also a button that you can depress the wheel to click. That is Button-3. From what I can tell, about 10% to 12% of people in the world are left hand dominate. They normally remap the buttons so that the “left click” button is the one on the right. When we use the terms “left click” and “right click”, it can be very confusing. So, start thinking in terms of Button-1 and Button-2 (and sometimes Button-3).

Important: The bindings editor (and many other editors in PAGE) does not support “normal” editing key functions like <Shift> Left Arrow or <Ctrl> Left Arrow. Always click at the end of the word(s) you want to edit or delete and use the <Backspace> key. It’s a Tcl/Tk thing. Sorry folks!

Once you have inserted your event, on the right side of the bindings window, you will see the binding code “`lambda e: xxx(e)`”. We’ll change the ‘`xxx(e)`’ part to ‘`btnGetText_lclick(e)`’ and then click the check mark on the right of the window to close it.

Note: You can also create the bindings in the `_support.py` module code. Many programmers find it better to define the binding when you are doing the GUI design.

We’ll do the same thing with the ‘**Print Checks**’ button, but point to a function called `btnPrintChecks_lclick(e)`.

Finally, bind the function ‘`btnExit_lclick(e)`’ to the `btnExit`.

TIP: You might wonder why we are using the wording “`lambda e:`” portion to create the binding. By using “`lambda e:`”, we are telling Python to not only create a callback for that widget, but to also send some extra information into the callback whenever the user of our program does something that cause the event.

Once again, save the .tcl file and generate the Python code and support module.

Step 3 – The Code

Now, we’ll write the code that will control what happens when our widgets are used.

Start by opening the `SecondApp_support.py` module in your favorite editor. You’ll see the standard Python import code, which we’ll ignore for now. Here is our skeleton support file.

```
def main(*args):
    '''Main entry point for the application.'''
    global root
    root = tk.Tk()
    root.protocol( 'WM_DELETE_WINDOW' , root.destroy)
    # Creates a toplevel widget.
    global _top1, _w1
    _top1 = root
    _w1 = SecondApp.Toplevel1(_top1)
    root.mainloop()

def btnExit_lclick(*args):
    print('SecondApp_support.btnExit_lclick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()

def btnGetChecks_lclick(*args):
    print('SecondApp_support.btnGetChecks_lclick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()

def btnGetText_lclick(*args):
    print('SecondApp_support.btnGetText_lclick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()

def on_rbClick(*args):
    print('SecondApp_support.on_rbClick')
    for arg in args:
        print('another arg:', arg)
```

```
sys.stdout.flush()
```

Here, just as a reminder, are the Tk variables that we set up when we used the PAGE designer. You can find these in the GUI.py file (SecondApp.py) around line 33.

```
self.ColorInfo = tk.StringVar()
self.ColorInfo.set('')
self.Colors = tk.IntVar()
self.che39 = tk.IntVar()
self.che40 = tk.IntVar()
self.che41 = tk.IntVar()
```

Remember, when we want to use these variables, since we are using them in the support file, we need to use **_w1**. as a prefix to reference the variable (and you don't need to include the **self**. prefix).

As you can see, everything is here for us. All we have to do is fill in a little bit of code. Let's start with modifying the main function so we can initialize our form before start up. All we are going to need is one line and maybe a comment. The line(s) you need to add will be in bold, while the rest of the function will be in non-bold. All functions from here out will be presented this way.

```
def main(*args):
    '''Main entry point for the application.'''
    global root
    root = tk.Tk()
    root.protocol('WM_DELETE_WINDOW', root.destroy)
    # Creates a toplevel widget.
    global _top1, _w1
    _top1 = root
    _w1 = SecondApp.Toplevel1(_top1)
    # My start up code
    initialize()
    root.mainloop()
```

Logically, we should next do the **initialize** function. We need to create the entire function this time.

```
def initialize():
```

```
# clear the checks from the checkbuttons
_w1.che39.set(0)
_w1.che40.set(0)
_w1.che41.set(0)
# set the first radiobutton (rbRed) to selected as default
_w1.Colors.set(1)
# Show which radiobutton is selected (rbRed)
on_rbClick()
```

The first thing we do is set all of the Checkbuttons to **zero (0)** by using the `.set()` method of the Checkbutton. Then, we set the default Radiobutton, which is the first Radiobutton widget with a **1** and then call the `on_rbClick` function to display the proper value in the label.

Let's do the Exit button function next. This will close and end our application correctly.

```
def btnExit_lclick(*args):
    print('SecondApp_support.btnExit_lclick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    root.destroy()
```

The `btnGetChecks_lclick` function will get the value of each Checkbutton then print to the console or terminal window a string that shows each of the checks. Since we are using a '**f-string**' to format the display string, be sure to use Python 3.7 or greater.

```
def btnGetChecks_lclick(*args):
    print('SecondApp_support.btnGetChecks_lclick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    chk1 = _w1.che39.get()
    chk2 = _w1.che40.get()
    chk3 = _w1.che41.get()
    # In this case, text is only local in scope
    text = ''
    if chk1:
```



```
        text = text + " Apples "
    if chk2:
        text = text + " Banana "
    if chk3:
        text = text + " Grapes "
    print(f'Checked item(s): {text}')
```

The **btnGetText_lclick** function will read the value that the user typed into the Entry widget and display it in the terminal.

```
def btnGetText_lclick(*args):
    print('SecondApp_support.btnGetText_lclick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    text = _w1.entryExample.get()
    print(f'Entry box contains: {text}')
```

Finally, the **on_rbClick** function will display the text of which Radiobutton was selected into the Label widget. We've created a list of the text that we want to display and then use the value of the Radiobutton that was clicked as an index into that list. Remember that Python list indexes are zero based, so we need to subtract **1** from that value to get the proper number as an index. So Red is index 0, Blue is index 1 and Green is index 2. When we set the values of each Radiobutton in PAGE, we used 1,2 and 3 respectively.

```
def on_rbClick(*args):
    print('SecondApp_support.on_rbClick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    which = _w1.Colors.get()
    print(f'Which: {which}')
    colornames = ['Red', 'Blue', 'Green']
    # Lists are zero based, so subtract 1 from the which variable
    _w1.ColorInfo.set(colornames[(which - 1)])
```

That's it.

When I ran our program, I did a very simple test of each of the functions. Here is what the terminal output looked like...

```
SecondApp_support.btnGetText_lclick
another arg: <ButtonPress event state=Mod2 num=1 x=29 y=15>
Entry box contains: blah
SecondApp_support.on_rbClick
Which: 1
SecondApp_support.on_rbClick
Which: 2
SecondApp_support.on_rbClick
Which: 3
SecondApp_support.btnGetChecks_lclick
another arg: <ButtonPress event state=Mod2 num=1 x=62 y=18>
Checked item(s): Apples Banana Grapes
SecondApp_support.btnExit_lclick
another arg: <ButtonPress event state=Mod2 num=1 x=71 y=15>
```

What you learned

By following this chapter, you have learned...

- How to create callbacks for Buttons and other widgets.
- How to bind callbacks using the Bind Editor.
- How to query various widgets with the **.get()** method.
- How to set various widget states with the **.set** method.
- How to use the **root.destroy()** method to close your application properly.

Good job. Take a break here. Chapters 3 and 4 are rather complicated. Fun, but complicated.

Chapter 3 – A more realistic project

As I said earlier, when I first started using Page, I wrote a tutorial for Full Circle Magazine, which Don was kind enough to make available from his website. That was back in 2012. Now that 2021 is here and Page has matured greatly, I decided to revisit that project and update it a bit.

What you will learn

The purpose of this project is to create a file seeker that will find audio and video files by their extensions that the user can select from a series of check buttons, then display the results in a spreadsheet like grid. This will demonstrate:

- Pop up dialog boxes, specifically the Directory Select dialog
- Proper use of Check buttons
- Use of the Scrolled Treeview widget
- The ability to change the cursor to a “busy” cursor and back.
- Filling an entry widget dynamically

This time, I won't give you the x,y position or the size of the widgets as I guide you through the creation of the UI. Instead, I will refer you to the image below and let you lay it out as you see fit. I will however, give you the important attributes that will match the code. Once you have the basic application working, please feel free to change it to match your needs. Here is a screen shot of the app GUI fresh out of Page.

Building the UI

We'll start by opening Page and move the new Toplevel designer widget to somewhere close to the middle of the screen both horizontally and vertically. You will also want to set the title and alias to “Searcher”. We'll need two standard frames about the same size, taking up somewhere around one half of the Topmost window vertically. Set the alias for the top one as **frameTop** and the bottom one to **frameTreeview**. You will want to save this in a new folder called Searcher and the .tcl project file should be named Searcher.tcl as well.

Now near the top of the upper frame, insert a label widget, entry widget and two button widgets and space them as you see in the above image. The label widget is simple to do, since it is a static text label. Set the text attribute to “**Path to search:**”. You can leave the alias set to the default of “Label1” if you wish, or change it to whatever suits you. That's it for the label widget.

Next we'll deal with our entry widget. Make it fairly long and somewhat higher than the default. Set the attributes as follows:

Attribute	Value
Alias	txtPaht
text variable	FilePath

Continuing to move along the widgets to the right, we'll deal with the first of the two button widgets. This one should be placed close to the right end of the entry widget and close to square. The text for the widget will only be three dots ('...'), meaning that if the user clicks this button it will bring up a dialog box of some sort. In this case, it will pop up a window that asks the user to select a directory to be searched. The file path that is returned will be entered into the entry widget programmatically by our code.

Attribute	Value
Alias	btnSearchPath
command	on_btnSearchPath
text	...

Our last button widget in this set will allow the user to exit the program.

Before we move on, set a binding for the **btnExit** to a function called **OnBtnExit(e)** to mouse button 1. Notice that on our previous button (**btnSearchPath**) we used the command attribute for setting the function, not a binding.

Attribute	Value
Alias	btnExit
text	Exit

Next we need two **LabelFrame** widgets to hold our check buttons in logical groups. Set their combined size to about $\frac{3}{4}$ of the horizontal space of frameTop. Make them both about the same size and along the same vertical (y) position so they line up. Set the label to the left one to “**Audio Files**” and the right one to “**Video Files**”. Add a button that will start the search process to the right of these **LabelFrames**. We'll come back in a few moments to set the attributes for this button.

Now we will need to put three check boxes into each LabelFrame. When I designed the UI, I put them close to the left side of the LabelFrame and aligned along the left side, to allow for additions in the future if needed.

TIP: The check buttons, by default, are set to center the text and the check box within the size of the widget. This can make it difficult to align the check boxes along a vertical line. We'll modify that by using the anchor attribute when we set up the definitions for each widget.

Starting with the Audio Files Labelframe, we'll put three **Checkbutton** widgets inside it. When I did the design, I spaced them with 30 pixels between them vertically, but you can lay them out however you wish. The three file types that we will be looking for when dealing with audio files are *.mp3, *.wav and *.ogg. This should take care of most of the audio files you might have on your computer. Again, you can add more later on if you want. Starting from the top and going down, the attributes are:

Attribute	Value
Alias	chkMP3
anchor	W
justify	Left
text	.MP3
variable	VchkMP3

Notice that I have changed the default variable name to “**Vchk**” plus the file extension. The 'V' stands for **Variable** and the **chk** stands for **CheckButton**. This will make our variable names much easier to remember and will make more sense when we start working with our code.

Note: A few words about variable and function naming conventions. If you are a professional programmer working for someone else, they will probably have a standard they want you to follow. If, however, you are like me and don't work for anyone, try to find a standard that you like and stick to it. I don't, but you should! Seriously, I've shown many different ways of naming functions and variables (and it's only Chapter 3!). Like my dear mother used to say, “Do as I say do, not as I do do.”

Attribute	Value
Alias	chkWAV
anchor	W
justify	Left
text	.WAV
variable	VchkWAV

Attribute	Value
-----------	-------

Alias	chkOGG
anchor	W
justify	Left
text	.OGG
variable	VchkOGG

Now in the Video Files LabelFrame, we'll repeat the process of placing three check buttons within the frame. They will allow the user to select from the following extensions for common video files, which are *.avi, *.mp4 and *.mkv. Again, I decided to place them close to the left side of the LabelFrame and stacked vertically with 30 pixel spacing between them to allow for future additions.

Attribute	Value
Alias	chkAVI
anchor	W
justify	Left
text	.AVI
variable	VchkAVI

Attribute	Value
Alias	chkMP4
anchor	W
justify	Left
text	.MP4
variable	VchkMP4

Attribute	Value
Alias	chkMKV
anchor	W
justify	Left
text	.MKV
variable	VchkMKV

Thinking about the future of the program, you could also add capabilities for looking for text files, PDF files, photos and who knows what else.

The last thing we need to do for the frameTop group is to work with the “go” button. Here are it's attributes:

Attribute	Value
Alias	btnGo
text	GO!

Finally bind the mouse button 1 to the button and point it to a function called **OnBtnGo(e)**.

Now we will work on frameTreeview. This part is fairly easy. We want to insert a ScrolledTreeview widget into frameTreeview. The ScrolledTreeview widget is near the bottom of the toolbox. Once it is in, move it to the upper left corner of the frame, then make it cover almost the entire frame. I usually like to leave a 2 to 4 pixel space of the frame showing to act like a border. You can leave to alias to the default of “Scrolledtreeview1”. The last thing you need to do here is to set a binding for the mouse button 1 to OnTreeviewClick(e). This will allow the user to select one of the files in the list.

We are using a Scrolled Treeview here to create a multi column list that holds the filenames and paths of the found files. We'll set all the configurations for the ScrolledTreeview in the code in a little bit.

Be sure to save your project and generate the code modules.

The Code

As always, you need to change the code in the support module to match the code presented here.

We need to make many changes the imports section of the code that PAGE generated for us to support some of the 'extra' things we want to do. In the support module, we need to include the **platform, os and os.path** library modules in addition to the **sys** module. By default, PAGE provides the import for the sys library.

```
import sys
import platform
import os
from os.path import join, getsize, exists
```

PAGE creates the basic Tkinter imports for us.

```
import tkinter as tk
import tkinter.ttk as ttk
from tkinter.constants import *
import searcher
```

We will be adding some extra Tkinter abilities, so we need to import some additional Tkinter modules.

```
from tkinter import messagebox
from tkinter import font
from tkinter import filedialog
```

The next part of the support module is the **main** function. Remember, we need to add code into the main function if we need to configure any widget options before the user sees our GUI. There are a number of things that we need to do right at the start, so we'll modify the main function to let us.

```
def main(*args):
    '''Main entry point for the application.'''
    global root
    root = tk.Tk()
    root.protocol('WM_DELETE_WINDOW', root.destroy)
    # Creates a toplevel widget.
    global _top1, _w1
    _top1 = root
    _w1 = searcher.searcher(_top1)
    init()
    root.mainloop()
```

As you can see, we will call the function **init**. You need to create the function as shown below.

```
def init():
    global treeview, exts, FileList
    exts = []
    FileList = []
    #-----
    BlankChecks()
    treeview = _w1.Scrolledtreeview1
    SetupTreeview()
    #-----
    global busyCursor, preBusyCursors, busyWidgets
```



```
busyCursor = 'watch'
preBusyCursors = None
busyWidgets = (root, )
```

We start by declaring three global variables, **treeview**, **exts** and **FileList**. **Ext**s and **FileList** we have already discussed. **treeview** is used to make it easier for us to refer in code to the ScrolledTreeview widget. We also set **exts** and **FileList** to empty lists. Next we call the **BlankChecks()** function to clear the checkbox widgets, set the global variable **treeview** to point to our Scrolledtreeview1 widget. Note that we use **w.Scrolledtreeview1**. The **'_w1.'** refers to our GUI and when we directly make calls to our widgets we have to prepend the **'_w1.'** Lastly, we setup the information for our busy cursor functions.

Now, we'll start defining the code that runs when we click one of the buttons or other widgets. First, we'll deal with a very simple one... the Exit button. Remember, we bound the mouse button-1 to a function named **OnBtnExit**. The first five lines within the function are created for us by PAGE. In order to exit the program, we use (as we have in the previous examples) the **root.destroy()** call.

```
def OnBtnExit(*args):
    print('searcher_support.OnBtnExit')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    root.destroy()
```

The **OnBtnGo()** function is where all the heavy lifting occurs. Once we have set the start folder and the extensions of the files we want to look for, we call all of the support functions here. Of course, we could have coded it to be one massive function, but it is much more readable in this format. First, clear the ScrolledTreeview then change the mouse cursor to “busy”. Build a list containing the file extensions we want to look for. Pull the file search path from the global variable and convert the extensions to a tuple. Clear the list containing the previously returned search results (if any) and call the recursive search function. Finally, load the search results into the grid and return the mouse cursor to the default state.

```
def OnBtnGo(*args):
    print('searcher_support.OnBtnGo')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    ClearDataGrid()
    busyStart()
    BuildExt()
```

```
fp = _w1.FilePath.get()
e1 = tuple(ests)
#Clear the list in case user wants to "go" again
del FileList[:] # under python 3.3, you can use list.clear()
Walkit(fp, e1)
LoadDataGrid()
busyEnd()
```

When the user clicks the '**get search path**' button (the one with the three dots), open a Tkinter askdirectory dialog to get the starting directory for the search. Rather than creating our own dialog box, we use one of the three built in file dialog pop-ups that are provided in the `tkFileDialog` module. The three include:

- `.askopenfile` – requests the selection of an existing file
- `.asksaveasfilename` – requests filename and directory to save or replace a file
- `.askdirectory` – requests a directory name.

While we or the user could simply enter the starting directory in the entry widget, this makes it easy for the user to enter the path properly. Once the dialog box is dismissed after selecting the starting directory, the path information is entered into the entry widget by means of the `.set(path)` method of the `FilePath` variable.

```
def OnBtnSearchPath(*args):
    print('searcher_support.OnBtnSearchPath')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    path = filedialog.askdirectory()
    _w1.FilePath.set(path)
```

The **OnTreeviewClick()** function is set up for later use.

```
def OnTreeviewClick(*args):
    print('searcher_support.OnTreeviewClick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    row = treeview.identify_row(args[0].y)
```

```
col = treeview.identify_column(args[0].x)
if row == None:
    print('Header Row')
elif row == '':
    print('Row Empty - Header Row')
print(f'Row: {row} - Col: {col}')
filename = treeview.set(row, 0)
path = treeview.set(row, 1)
if row != '':
    titl = 'Searcher Treeview Click'
    msg = f'Filename: {filename} - Path: {path}'
    messagebox.showinfo(title=titl, message=msg)
```

When we start the program, we want all of the checkbutton widgets to be unchecked. We use the **BlankChecks()** function to set the variables associated with each of the Checkbuttons to a 0 or unchecked state.

```
def BlankChecks():
    _w1.VchkAVI.set(0)
    _w1.VchkMKV.set(0)
    _w1.VchkMP3.set(0)
    _w1.VchkMP4.set(0)
    _w1.VchkOGG.set(0)
    _w1.VchkWAV.set(0)
```

The **BuildExts()** function creates a list of extensions that will be used by the recursive filename function (Walkit()). We simply check the variable associated with each of our Checkbox widgets to see if it has a value of 1. If so, we append that extension text to the list called 'exts'. When we enter the function, we want to empty the list, just in case the user changes the an extension choice and runs through the process again. We do that by using the 'del list[:]' command. Python3 also allows a list.clear() method that could replace this one.

```
def BuildExts():
    del exts[:] # Clear the extentions list, then rebuild it...
    if _w1.VchkAVI.get() == 1:
        exts.append(".avi")
    if _w1.VchkMKV.get() == 1:
```

```
        exts.append(".mkv")
    if _w1.VchkMP3.get() == 1:
        exts.append(".mp3")
    if _w1.VchkMP4.get() == 1:
        exts.append(".mp4")
    if _w1.VchkOGG.get() == 1:
        exts.append(".ogg")
    if _w1.VchkWAV.get() == 1:
        exts.append(".wav")
```

The **busyStart()** function changes the mouse cursor to a 'watch' cursor for the root window and all of the child widgets, with the exception of the entry widget. This works well under Linux, but the Windows OS has an issue of doing this while the recursive filename scan runs.

```
def busyStart(newcursor=None):
    global preBusyCursors, busyWidgets
    print('busyStart')
    if not newcursor:
        newcursor = busyCursor
    newPreBusyCursors = {}
    for component in busyWidgets:
        newPreBusyCursors[component] = component['cursor']
        component.configure(cursor=newcursor)
        component.update_idletasks()
    preBusyCursors = (newPreBusyCursors, preBusyCursors)
```

The **busyEnd()** function resets the mouse cursor back to the default cursor.

```
def busyEnd():
    global preBusyCursors, busyWidgets
    print('busyEnd')
    if not preBusyCursors:
        return
    oldPreBusyCursors = preBusyCursors[0]
```

```
preBusyCursors = preBusyCursors[1]
for component in busyWidgets:
    try:
        component.configure(cursor=oldPreBusyCursors[component])
    except KeyError:
        pass
    component.update_idletasks()
```

Here is the real heart of our program. The **Walkit()** function takes the starting folder and the list of extensions, converted to a tuple, and recursively walks down looking for a file that has an extension that matches one that we are looking for. If a file is found that matches, its filename and path are added to another list (**fl**) which is then added to the list to be returned. It continues this until all files under all folders below the start folder have been checked.

```
def Walkit(musicpath, extensions):
    print(f'Into Walkit - Path: {musicpath} - Exts: {extensions}')
    rcntr = 0
    fl = []
    for root, dirs, files in os.walk(musicpath):
        rcntr += 1 # This is the number of folders we have walked
        for file in [f for f in files if f.endswith(extensions)]:
            fl.append(file)
            fl.append(root)
        FileList.append(fl)
        fl = []
```

The **SetupTreeview()** function will set the number of columns and the headers for each column from the global list **ColHeads** using the `.configure` method. In this case, we will have just two columns named 'Filename' and 'Path'.

Note: that the *SetupTreeview* function must be called before attempting to load data into the grid.

```
def SetupTreeview():
    global ColHeads
    ColHeads = ['Filename', 'Path']
```

```
treeview.configure(columns=ColHeads, show="headings")
for col in ColHeads:
    treeview.heading(col,
                      text=col.title(),
                      command=lambda c=col: sortby(treeview, c, 0))

    ## adjust the column's width to the header string
    treeview.column(col, width=font.Font().measure(col.title()))
```

The **ClearDataGrid()** function will remove all data from the Scrolledtreeview widget. If you are going to use the Scrolledtreeview in other projects, this would be a handy one to keep in your tool kit. Basically, all it does is walk through the treeview widget and deletes each of the data items one by one.

```
def ClearDataGrid():
    #print("Into ClearDataGrid")
    for c in treeview.get_children(''):
        treeview.delete(c)
```

The **LoadDataGrid()** function first clears the treeview widget and then takes the returned results from the **Walkit()** function and loads each result into a row in the Treeview widget. The remainder of the code will readjust the column widths to fit the longest value.

```
def LoadDataGrid():
    global ColHeads
    ClearDataGrid()
    for c in FileList:
        treeview.insert('', 'end', values=c)
        # adjust column's width if necessary to fit each value
        for ix, val in enumerate(c):
            col_w = font.Font().measure(val)
            if treeview.column(ColHeads[ix], width=None) < col_w:
                treeview.column(ColHeads[ix], width=col_w)
```

Finally, we have a function called **sortby()** that, when a column header is clicked, will sort the Treeview. This is bound to the ScrolledTreeview in the SetupTreeview() function.

```
def sortby(tree, col, descending):
```

```
# sort tree contents when a column header is clicked on
# grab values to sort
data = [(tree.set(child, col), child) \
        for child in tree.get_children('')]
# if the data to be sorted is numeric change to float
#data = change_numeric(data)
# now sort the data in place
data.sort(reverse=descending)
for ix, item in enumerate(data):
    tree.move(item[1], '', ix)
# switch the heading so it will sort in the opposite direction
tree.heading(col, command=lambda col=col: sortby(tree, col, \
        int(not descending)))
```

When you run the program, you can click on an entry in the Treeview, but nothing will happen, excepting a short print to the terminal window. While this can be a helpful program for finding specific media files in your library, wouldn't it be nice if you could play a selected file, audio or video, by simply running this program and then clicking on it in the treeview grid? We'll do that in Chapter 4.

What you have learned

This chapter has given you a large amount of extra knowledge.

- Using Pop up dialog boxes, specifically the Directory Select dialog.
- How to use the Scrolled Treeview widget.
- How to change the cursor to a “busy” cursor and back in code.
- How to Fill an entry widget dynamically.

Take another break. Chapter 4 is a long one with lots of form design and code.

Chapter 4 – NASA Still Image Viewer

You’ve learned a large amount about PAGE and we will add a great amount to your learning in this project. This time we will learn how to create a front end GUI to display still images that are downloaded from NASA.

I originally wrote this as an article for Full Circle Magazine (FCM #175 - HowTo - Python in the REAL World - Part 123) using PAGE 6.2. This chapter is simply an updated version of that article with the focus on using PAGE 7 rather than PAGE 6.2. I strongly suggest you look at both, to see how easy it is to convert the code for a project from PAGE 6 (or before) to PAGE 7.

Pictures Pictures Everywhere

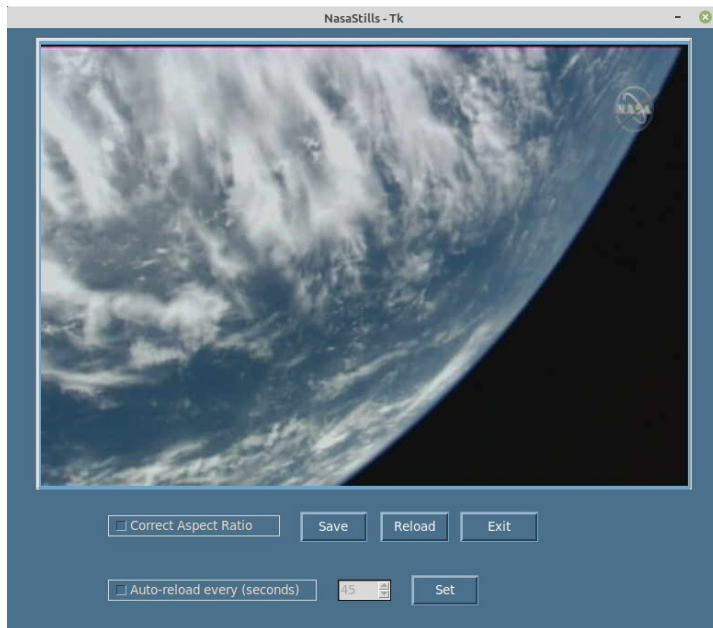
A few weeks ago, one of my Python newsfeeds alerted me about a neat new project – one that creates a still video image that is downloaded from NASA (National Aeronautics and Space Administration) every 45 seconds or so and displays it in the application’s GUI window.

I thought that was a neat idea, so I jumped to the website and started reading. It seems that the author decided to use PySimpleGUI to make the GUI form, which is a program that I’m not really familiar with. I took a look at his screenshot of the GUI and I thought that I could create a comparable version using PAGE. I took up the challenge and thought that it would be a good project to present here. So I present my version of Spacestills.py.

The author’s blog is at <https://blog.paoloamoroso.com/2021/04/a-nasa-tv-still-frame-viewer-in-python.html>, and is nicely presented. His source code can be found at <https://github.com/pamoroso/spacestills>.

We’ll be using two external Python libraries. You might already have them. They are requests and PIL (pillow). If you don’t have them, you can use pip to install them. They have to be on your system before you try to run the project.

Here is what my version of his project looks like during a running session.



I decided to make a little bit more space between the rows of user accessible widgets, since my mousing hand has a tendency to shake a bit. You should use Python 3.6 or higher, since I use f-strings for some of the print statements.

Let's look at some of the requirements that we need to keep in mind during the design process. The images that come in from the Internet will be 704x480 pixels. The shortest time between the images is 45 seconds. This information came from the author's blog page as well as his code (his blog page states the image size is 704x408, but the actual image and his code is 704x480). The only code of his that I used are a few of his constants. The rest of the code for this project is code that I have used in previous projects. His project allows the end user to select the time between requests to be between 45 seconds and 300 seconds. He also allows the ability to change the aspect ratio from 704x480 to a 16x9 format which means that the image will be resized to 704x396 before it is displayed and/or saved. There is a Checkbutton that allows for this resize to be performed. The author has a Save button that allows the image to be saved to the hard drive, as well as a Reload button. This is helpful if you change between the native image size to the 16x9 format (or if you are impatient and don't want to wait for the next refresh to occur). He also provides a Checkbutton to change the delay time between requests, and a text entry widget (and a button) to set the new time. I decided to use a Tk::Spinbox to make the time selection easier and doesn't require the Set button. I do, however, include the set button, but all it does is bring up a warning Message box. The reason that I decided to use the Spinbox is that if you use the Up/Down arrows on the Spinbox, you don't have to check if the value is numeric, keeping the code simpler.

What you will learn

Some of the things that you'll learn in this chapter are:

- Dynamically display images in the GUI
- Use a Spinbox widget to control program variables
- The '.after' method
- Fully event driven programming

Getting Started

You will need to make sure that you have two additional Python libraries to get the program to work. They are requests and Pillow. You can use pip to install them. Open a terminal and type

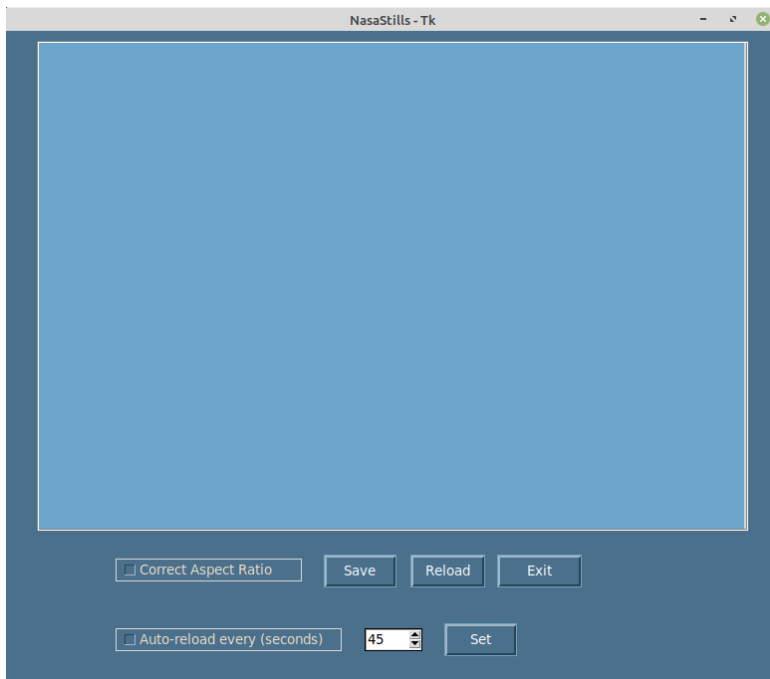
```
pip install requests
```

```
pip install pillow
```

Now we can concentrate on building the GUI. This is actually very easy for this project.

Building the GUI

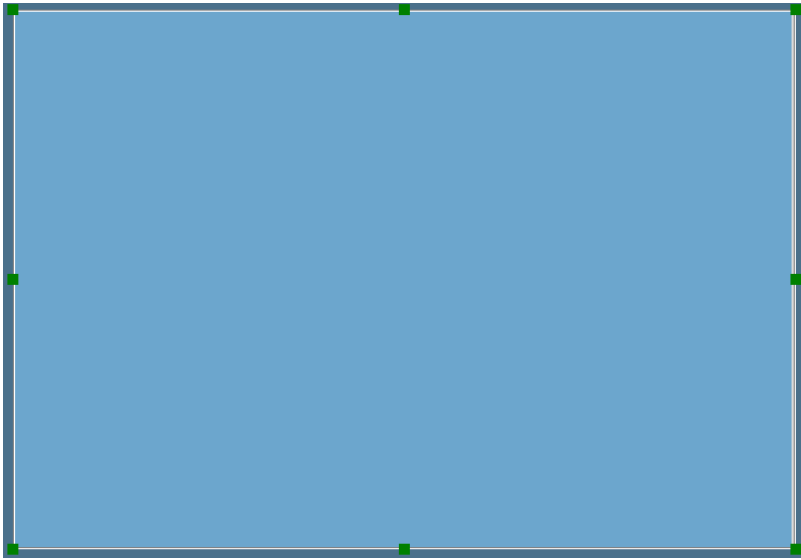
This is what our GUI looks like when the design process is finished.



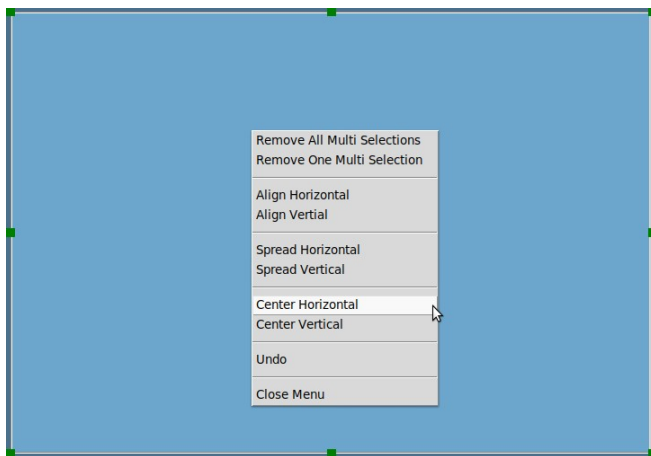
So start up PAGE, and resize the default Toplevel designer window to about 777 pixels wide and 657 high. Don't be too concerned about the actual dimensions of the main form, just get it close. Now move the form to somewhere in the middle of your screen. Set the title attribute in the Attribute Editor to "**NasaStills-Tk**". While we are in the Attribute Editor, set the background color to "skyblue4". This is a nice dark bluish gray color.

Next, we'll put a frame widget into the Toplevel form. This will hold the Label widget that we will use to display the still image from the Nasa site. Place it near the top-left of the main form. Don't worry too much about the placement at this point. Using the Attribute Editor again, make sure that the frame is selected, and set the width to 714 and the height to 492. Next, add a Label widget and set the X and Y position in the Attribute editor to 2 and 2. This will provide a nice little border around the Label widget when it is set to its full size. Set the width of the Label to 704, the height to 488, and the background color to "skyblue3". This gives us a nice color for the visual inner display area. Set the Alias of the Label widget to "labelImage", and delete the text in the text attribute.

Now we will finish the placement of the Frame and Label combo. PAGE has a feature that allows you to select multiple widgets and manipulate them as a group, like centering horizontally or vertically, equally spacing them within their parent, and so on. At this point, we want to just center the frame within the main form horizontally. In the Widget Tree, use the middle mouse button to click on the Frame: Frame1 entry.



Notice that the label in the Widget Tree turns green. The black resize handles of the Frame in the main form also turn green to let you know you are in the multi-select mode. We need to select only the Frame, since it is the parent of the image label. Now right-click on the Frame/Image combo in the main form. You will see a context menu appear.



Select Center Horizontal from the menu. The Frame will now be centered in the form horizontally. Click Unselect MS in the Widget Tree window and the resize handles turn back to black. I set the top of my Frame to 10 pixels down from the top of the form (Y position). It's time to save your project. Name it "NasaStills.tcl".

Now we can start to place the rest of the widgets on our form. We'll place the first of two Checkbuttons next. Be sure to make it a Tk Checkbutton, not a ttk Themed Checkbutton.

I'll give you the attributes you will want to set in the grid below.

Attribute	Value
Alias	chkAspect
active bg	skyblue3
anchor	w
background	skyblue4
command	on_chkAspect
foreground	antiquewhite2
select color	skyblue4
text	Correct Aspect Ratio
variable	che47

Some information about these attributes. By setting the Active Background (active bg) and select color attributes, we control the colors when the mouse is over the widget (active bg) as well as the color when the status is Checked (select color). Setting the foreground color to antiquewhite2, the text and check area are bright enough to be seen, but not bright enough to cause the check not to show. If we were to set the foreground to full white (#ffffff), you won't be able to tell when the check is set. Give it a try.

Save again and we'll add three more buttons. Make sure they line up nicely in the row with the Checkbutton. The first one will be the Save button, next is the Reload button, and the last is the Exit button. You might want to sneak a peek at the image above for a reference. Here are the attributes for the Save button.

Attribute	Value
Alias	btnSave
active bg	skyblue3
background	skyblue4
command	on_btnSave
foreground	#ffffff

highlight bg	skyblue3
text	Save

Next we'll do the Reload button

Attribute	Value
Alias	btnReload
active bg	skyblue3
background	skyblue4
command	on_btnReload
foreground	#ffffff
highlight bg	skyblue3
text	Reload

Finally the Exit button.

Attribute	Value
Alias	btnExit
active bg	skyblue3
background	skyblue4
command	on_btnExit
foreground	#ffffff
highlight bg	skyblue3
text	Exit

Remember to make everything line up in the row nicely, and provide a little bit of space between the bottom of the frame and the top of the buttons.

Save your project again and we'll finish up with the last three widgets. First we'll do another Tk Checkbutton.

Attribute	Value
Alias	chkTime
active bg	skyblue3
anchor	w
background	skyblue4
command	on_chkTime
foreground	antiquewhite2
select color	skyblue4
text	Auto-reload every (seconds)
variable	che48

Next we will place the Spinbox. You'll want to make it a bit smaller in width from the default that PAGE gives you. I made mine only 58 pixels wide. How wide you make it is up to you. Here are the attributes.

Attribute	Value
Alias	Spinbox1
from	45.0
to	300.0
command	on_spinChange
text var	spinbox
wrap	Yes

Last but not least, is the Set button. The original author used this to apply the time delay from an Entry style widget. I'm going to take advantage of the Spinbox **text var** to update the delay time every time we get an image. So if you want to include it, that's fine. If not, that's not a problem.

Attribute	Value
Alias	btnSet
active bg	skyblue3
background	skyblue4
command	on_btnSet
foreground	#ffffff
highlight bg	skyblue3
text	Set

We are all done with the design form. Save your project and generate your Python modules. Next we'll work on the code in the `_support` module.

Take a quick break. You deserve it!

The Code

We'll start with the imports section. Unlike most of our previous projects in this tutorial, the imports section for this project is fairly long. The lines below in bold need to be added.

```
import sys
from os.path import exists
from datetime import datetime, timedelta
from pathlib import Path
import requests
from requests.exceptions import Timeout
```



```
from PIL import Image, ImageTk
import shutil

import tkinter as tk
import tkinter.ttk as ttk
from tkinter.constants import *

from tkinter import Spinbox, messagebox
from tkinter import font
from tkinter import filedialog
import Chapter4
```

In the version 6.2 version, I talked about modifying the `set_Tk_var` and `init` functions. However, as you know by now, these two functions don't exist in a PAGE 7 project. We'll simply add a single command to the **main** function and then do the rest in the startup function that we will add.

```
def main(*args):
    '''Main entry point for the application.'''
    global root
    root = tk.Tk()
    root.protocol('WM_DELETE_WINDOW', root.destroy)
    # Creates a toplevel widget.
    global _top1, _w1
    _top1 = root
    _w1 = Chapter4.Toplevellevel1(_top1)
    startup()
    root.mainloop()

def startup():
    global feed_url, WIDTH, HEIGHT, HEIGHT_16_9, MIN_DELTA, MAX_DELTA, DELTA
    global Timer_ID, resize, refresh_time, debug
    feed_url = 'https://science.ksc.nasa.gov/shuttle/countdown/video/chan2large.jpg'
    # Frame size without and with 16:9 aspect ratio correction
    WIDTH = 704
    HEIGHT = 480
```

```
HEIGHT_16_9 = 396
# Minimum, default, and maximum autoreload interval in seconds
MIN_DELTA = 45
DELTA = MIN_DELTA
MAX_DELTA = 300
refresh_time = MIN_DELTA
resize = True
_w1.spinbox.set(DELTA)
_w1.ch48.set(0)
_w1.ch47.set(0)
_w1.Spinbox1.configure(state=DISABLED)
debug = True
# check for existence of the save file counter
if exists('filecounter.txt'):
    pass
else:
    filecount = open("filecounter.txt", "w")
    filecount.write('0')
    filecount.close
# Set up the root.after timer
centre_screen(777, 657)
Timer_ID = root.after(0, on_tick())
```

Here's the code for our Exit button callback.

```
def on_btnExit(*args):
    print('Chapter4_support.on_btnExit')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    root.destroy()
```

The Reload button callback is very simple as well. We just call the **get_image_from_web** function.

```
def on_btnReload(*args):
    print('Chapter4_support.on_btnReload')
    for arg in args:
```

```
        print('another arg:', arg)
    sys.stdout.flush()

    global feed_url
    get_image_from_web(feed_url)
```

The Save button callback is a little bit more complicated, but it's not too bad. It will attempt to save the current image to a file. It first opens the filecounter.txt file (which is created at startup if it doesn't exist), reads the number of the last file (0 if none have been saved), increments by one, and then appends that value to "NasaStills" in the filename "NasaStills?.png". All files are saved directly to the source code folder.

```
def on_btnSave(*args):
    print('Chapter4_support.on_btnSave')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()

    # get last filename
    filecount = open('filecounter.txt', 'r+')
    global lastfile
    lastfile = filecount.read()
    lastfilename = int(lastfile) + 1
    filecount.seek(0)
    filecount.write(str(lastfilename))
    filecount.close

    src = 'local_image.png'
    dst = f'NasaStills{lastfilename}.png'
    shutil.copyfile(src, dst)
```

The on_btnSet callback function is not needed if you decided not to include the Set button. If you did, and followed the instructions, this will cause an information dialog box to be displayed by calling the showinfo function.

```
def on_btnSet(*args):
    print('Chapter4_support.on_btnSet')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
```

```
showinfo("Set", "Set function is not yet implemented")
```

The callback for the Checkbutton `chkAspect` really isn't needed. I included it just to be complete and to re-enforce the use of the callback for the Checkbutton.

```
def on_chkAspect(*args):
    print('Chapter4_support.on_chkAspect')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
```

Here is the **chkTime** function I told you about. We check if the Checkbox variable is equal to 1. If so, we set the **spinbox** state to normal. Otherwise set it to disabled.

```
def on_chkTime(*args):
    print('Chapter4_support.on_chkTime')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if _w1.chk48.get() == 1:
        _w1.Spinbox1.configure(state=tk.NORMAL)
    else:
        _w1.Spinbox1.configure(state=tk.DISABLED)
    refresh_time = _w1.spinbox.get()
    root.update()
```

The **on_spinChange** callback function fires every time the **spinbox** is incremented or decremented. It simply sets the global variable `refresh_time`. It's an artifact from an earlier version that I did, since the **on_tick** timer function currently gets the value directly from the **spinbox** to set the next timer value. I provided it here just as an example of how to get the value of the **spinbox**.

```
def on_spinChange(*args):
    print('Chapter4_support.on_spinChange')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    global refresh_time
    refresh_time = _w1.spinbox.get()
```

The `get_image_from_web` function is where the first part of the “magic” happens. We use the **requests.get** method from the requests library to grab an image from a website. The image is received and saved as a .png image. Then, we verify the width and height of the image. If it is the correct size (704x480), and the aspect ratio Checkbutton is not checked, then it is simply saved as a local image. If the aspect ratio Checkbutton IS checked, then the **PIL** library resizes to the 16x9 format before it gets saved. Finally, we set the image attribute of the Label to the image we just downloaded. We also save it to a convenient local file, in case the user wants to save it.

```
def get_image_from_web(url):
    # Attempt to get image from url and place it in w.lblImage
    global _img2, debug
    pic_url = url
    if debug:
        print(f'Refresh Time: {refresh_time}')
        print(f'Attempting to get {url}')
    try:
        with open('pic1.jpg', 'wb') as handle:
            response = requests.get(url, stream=True)
            if not response.ok:
                print(response)
            for block in response.iter_content(1024):
                if not block:
                    break
                handle.write(block)
        jpgfile = Image.open('pic1.jpg')
        jpgfile.save('local_image.png')
        original = Image.open('local_image.png')
        wid, hei = original.size
        if _w1.che47.get():
            newheight = HEIGHT_16_9
        else:
            newheight = HEIGHT
        if debug:
            print(f'Width: {wid} - Height: {hei}')
        _img1 = original.resize((WIDTH, newheight), Image.ANTIALIAS)
        _img2 = ImageTk.PhotoImage(_img1)
```

```
_w1.labelImage.configure(image=_img2)
_img1.save('local_image.png')
except Exception:
    boxTitle = "Image Error"
    boxMessage = "An error occurred getting the image."
    showerror(boxTitle, boxMessage)
    if debug:
        print("An error occurred getting the image")
    _img2 = None
    _w1.labelImage.configure(image=_img2)
```

The **on_tick** function is where the rest of the “magic” happens, in my mind at least. Here we use the `root.after` function of Tkinter to create a timer event. The first thing we do is to get the current value of the spinbox (between 45 and 300) and store it in a temporary variable `rt` (standing for refresh time). It then calls the `get_image_from_web` function to get the image and refresh the `imageLabel`. Finally the callback is enabled with the time in milliseconds (`rt * 1000`) and resets the callback for the next call.

```
def on_tick():
    global Timer_ID, debug
    global feed_url # , WIDTH, HEIGHT, HEIGHT_16_9, MIN_DELTA, MAX_DELTA, DELTA
    if debug:
        print('Into on_tick')
        print(f'RefreshTime = {_w1.spinbox.get()}')
        print(datetime.now())
    rt = int(_w1.spinbox.get())
    if debug:
        print(f'rt: {_w1.spinbox.get()}')
    get_image_from_web(feed_url)
    Timer_ID = root.after(rt * 1000, on_tick)
```

The **showinfo** function takes two parameters, title and message, then calls the Tkinter message box to show the message box to the user. We also provide the parent (which in this case will always be root, but that will keep the messagebox over the actual application and set the icon to the INFO icon.

```
def showinfo(titl, msg):
```

```
messagebox.showinfo(titl, msg, parent=root, icon=messagebox.INFO)
```

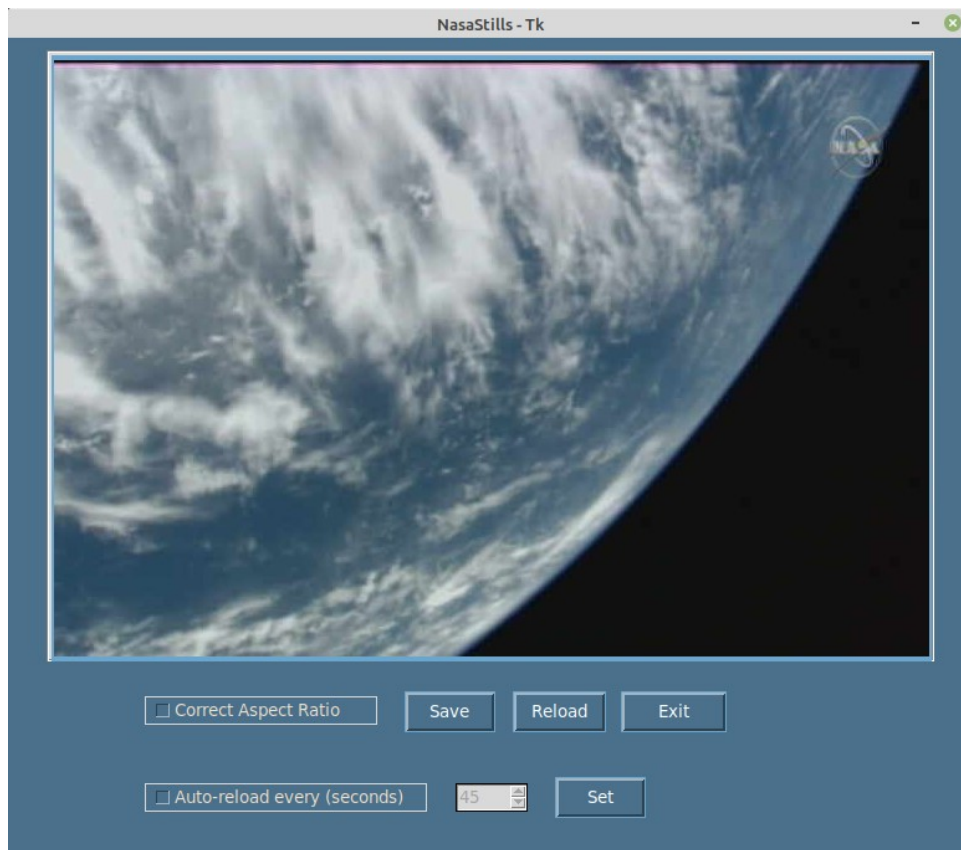
The **showerror** function is almost exactly the same as the showinfo function, but shows an error message box with an error icon instead.

```
def showerror(titl, msg):  
    messagebox.showerror(titl, msg, parent=root, icon=messagebox.ERROR)
```

Finally, we have the **centre_screen** function, which takes the width and height of our main form (which is provided in the GUI python file that PAGE creates). It then uses the screen width and height to calculate the centre of the screen and form.

```
def centre_screen(wid, hei):  
    ws = _top1.wininfo_screenwidth()  
    hs = _top1.wininfo_screenheight()  
    x = (ws / 2) - (wid / 2)  
    y = (hs / 2) - (hei / 2)  
    _top1.geometry('%dx%d+%d+%d' % (wid, hei, x, y))
```

When we run the program, this is what it should look like



Of course, the chances of the image being the same are slim to none, but you should see some sort of image right away.

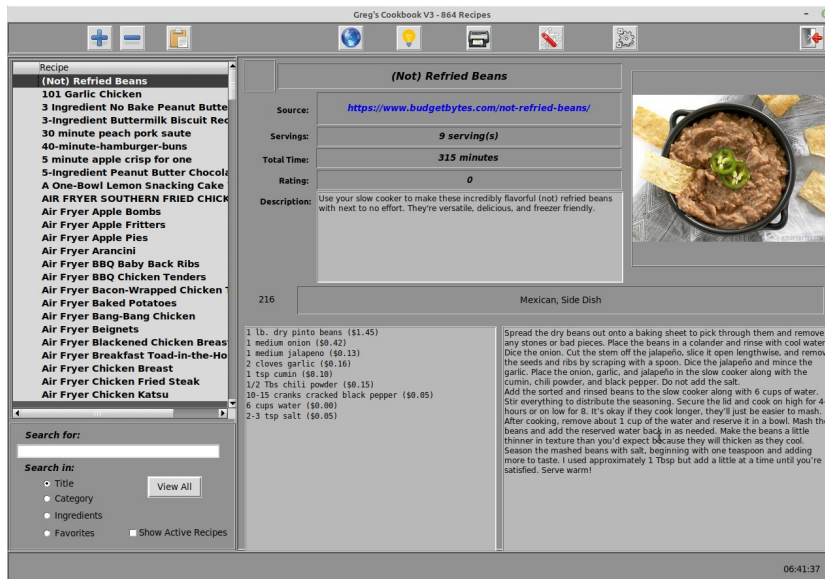
That's it. Our project is completed. I feel happy about the functionality that Tkinter provides against the PySimpleGUI toolkit. When I did the first version of the program, it took me less than 30 minutes to design the PAGE GUI form and write or borrow the code functions from some of my previous programs. The length of the actual support module with comments (LOTS of comments), and double spacing between functions, is only about 230 lines of code, which isn't too bad.

We have one more project to go. Chapter 5 will talk about using Menus and Multiple forms.

Chapter 5 – Multiple Form Applications

Since the original tutorial included a chapter on how to create applications that use multiple forms and since one of the biggest feature changes of PAGE 7 is using Multiple Forms, I decided to include a chapter on this topic.

First, let's discuss what a Multiple Form project actually is. As the name suggests, a Multiple Form project has multiple Toplevel forms. You, of course will have a main form, but there are other Toplevel forms that can be visible at all times, and/or Toplevel forms that show up only when needed and then go away. PAGE itself is a multiple form application. Take for example a program that I wrote for myself that keeps track of recipes. The main form looks like this...



It allows me to search for recipes in the internal database, see what the recipe should look like when finished, the list of ingredients, the instructions, where the recipe came from and so on. There is another form that allows me to, once I've found a recipe that I want to add to the database from an online recipe site, potentially scrape the data from the site and add it to the database.

Learning PAGE - A Python GUI Designer

The screenshot shows a window titled "Greg's Recipe Website Scraper". It contains a "Website:" field with the URL "https://www.allrecipes.com/recipe/256129/smittys-low-carb-chili/". A "Scrape" button is next to it. Below this, the "Title:" field shows "Smitty's Low-Carb Chili". To the right of the title is a "Save To Database" button. The "Total Time:" field shows "48 minutes" and the "Yields:" field shows "4 servings". The "Image URL:" field shows a long URL starting with "https://imagesvc.meredithcorp.io/v3/mm/image?url=https%3A%2F%2Fimages.media-allrecipes.com%2Fuserphotos%2F9376165.jpg". Below the image URL is a "Description:" field with text about finding variety on a low-carb diet. To the right of the description is a small image of a bowl of chili. Below the description is a "Categories:" section with a list of categories including Grains, Greek, Hamburger, Holiday, Indian, Instant Pot, Italian, Japanese, Latin American (checked), Main Dish (checked), Marinade, and Meats. To the right of the categories is an "Ingredients:" section with a list of ingredients including cauliflower, butter, ground beef, pork loin, onion, green pepper, tomatoes, smoked sausage, chili seasoning mix, beef stock, hot sauce, salt, black pepper, and cheddar cheese. To the right of the ingredients is an "Instructions:" section with a list of steps for making the chili. At the top right of the window is an "Exit" button.

This form only shows when I want it to. Since some websites don't work with the scraper, or when I want to create a recipe from scratch, I've got a simple editor form that can be shown as either a new recipe entry form or as a editor form for existing recipes.

The screenshot shows a window titled "Editor v0.5.6 - Edit Mode". It contains a "Record Number:" field with the value "126". A "Save" button is next to it. Below this, the "Title:" field shows "Air Fryer Apple Fritters". To the right of the title is a "Save" button. The "Source:" field shows the URL "https://www.allrecipes.com/recipe/274396/air-fryer-apple-fritters/". The "Servings:" field shows "4 serving(s)" and the "Total Time:" field shows "25 minutes". The "Rating:" field shows "3 1/2". Below the rating is a "Description/Notes:" field with text about making apple fritters in an air fryer. To the right of the description is a small image of apple fritters. Below the description is a "Categories:" section with a list of categories including Adult Beverage, African, Air Fryer (checked), American, Appetizer, Asian, Barbecue, Beans, Beef, Beverages, Breads, and Breakfast (checked). To the right of the categories is an "Ingredients:" section with a list of ingredients including cooking spray, all-purpose flour, white sugar, milk, egg, baking powder, salt, white sugar, ground cinnamon, peeled/cored/chopped apple, confectioners' sugar, milk, caramel extract (such as Watkins'), and ground cinnamon. To the right of the ingredients is an "Instructions:" section with a list of steps for making the fritters. At the top right of the window is an "Exit" button.

Again, this form only shows when needed. Under PAGE 6.x and before, each of the forms needed to be created as a standalone project and the files saved as such. This meant that I had to have three files for each form, not including any image files, database files or configuration files.

PAGE 7 changes all of that. Now we create each form as part of a project and all information about the forms are all stored in a single project, just three files. You still have to manage the image files, database files and so on, but this greatly cuts down on the number of files you have to worry about.

What you will learn

Here are just some of the things that you will learn in this chapter.

- Creating a Form level menu
- Creating a Button Bar Menu
- Creating multiple forms
- Controlling which form is currently showing

Creating the GUI

As we always do when we start a new PAGE project, start PAGE. Click on the Toplevel form and set the Alias to “Main” and set the title to “Main Form”. While it’s always a good idea to set the Alias of the Toplevel, it is extra important to set the Alias for all the forms in a multiple form application. You’ll see why in a few minutes, but for now, just believe me.

Save your file now as **Multiple.tcl**.

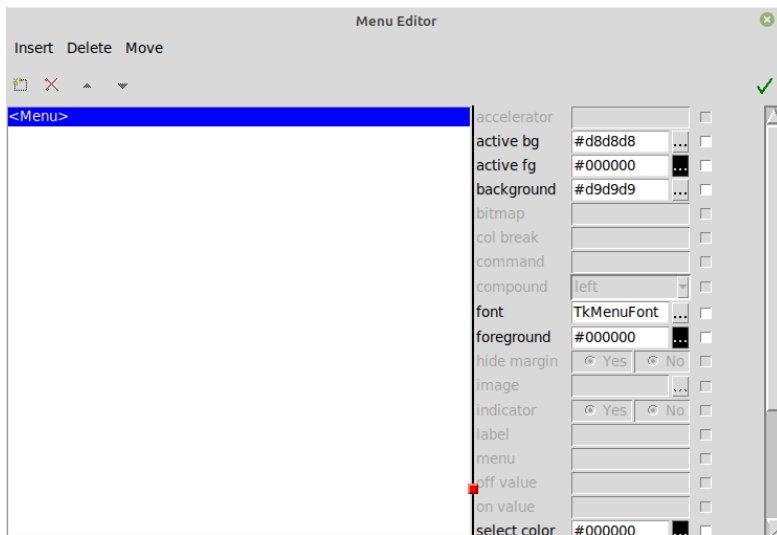
Now we’ll create a form level menu. This is a standard text based menu that you are probably familiar with. It will be a very simple menu, but this will give you the basics of creating menus for your applications. While many applications don’t need menus like the previous projects in this tutorial, there are times that you will want a Form level menu (textual, like the PAGE main form) or a graphical Button Bar menu like my recipe main form. We’ll create both kinds of menus.

Creating the Form Level Menu

When creating a Form level menu, it’s best to do it right from the start. That way, it’s already there when you design your form layout. To create the menu, PAGE provides a nice Menu Editor. You access it from the Attribute Editor.

highlight bd	0	<input type="checkbox"/>
menu	<click to edit>	<input type="checkbox"/>
x pad	0	<input type="checkbox"/>

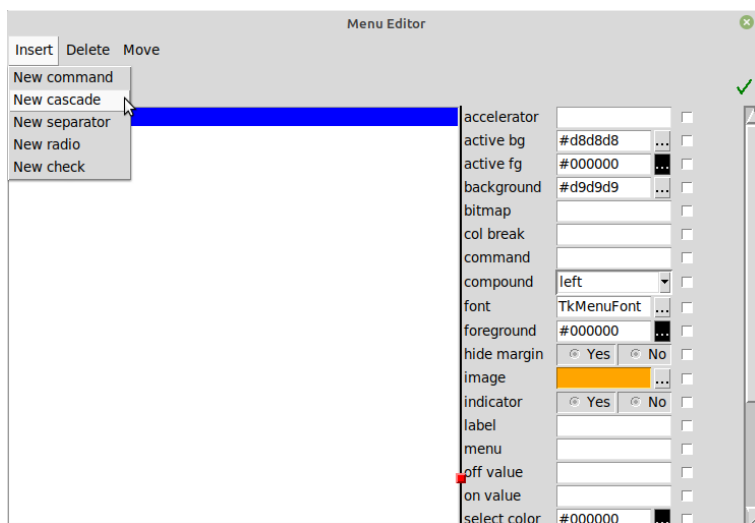
As the prompt says, just click on the <click to edit> box in the Attribute editor. This will bring up the Menu Editor form.



Notice that the Menu Editor has the two types of menus itself. The textual and a button bar style. You can use either. The options are the same for both. From the left to the right the options are Insert, Delete and Move (Move Up and Move Down for the button bar). Insert will add a menu item, Delete will delete the selected item and the move options will move the selected item up and down the menu tree. (We'll see more about the menu tree in a moment.)

You have to Insert each menu items one at a time. This will create an item in the tree panel in the lower left part of the editor. The tree, as you would expect will grow as you add more items.

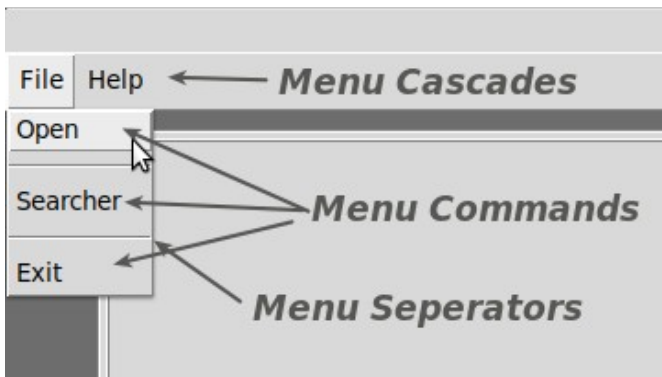
When you click on Insert (either the Textual version or the button version) you will see a context menu that gives you the allowable options.



Your options are:

1. New command
2. New cascade
3. New separator
4. New radio
5. New check

There can be some confusion as to what the difference between the New Command and the New Cascade options are. The Separator, Radio and Check are fairly self explanatory. I've created an annotated image to help you understand.



As you can see, the Cascades are “Top” level menu items. They are visible at all times as long as the Menu itself is visible. Think of them as Command Groupings. The Commands are just that, items that do a job. You can create menus that have Cascades that have Cascades that have Commands. As many levels as you wish. That having been said, spend some time thinking out your menu before you start creating it. The easier it is for a user of your program to navigate the menu to find what they are looking for, the better they will like it.

I thought about our menu for a while, trying to figure out the best options and layout for it and came up with this textual representation.

Actions

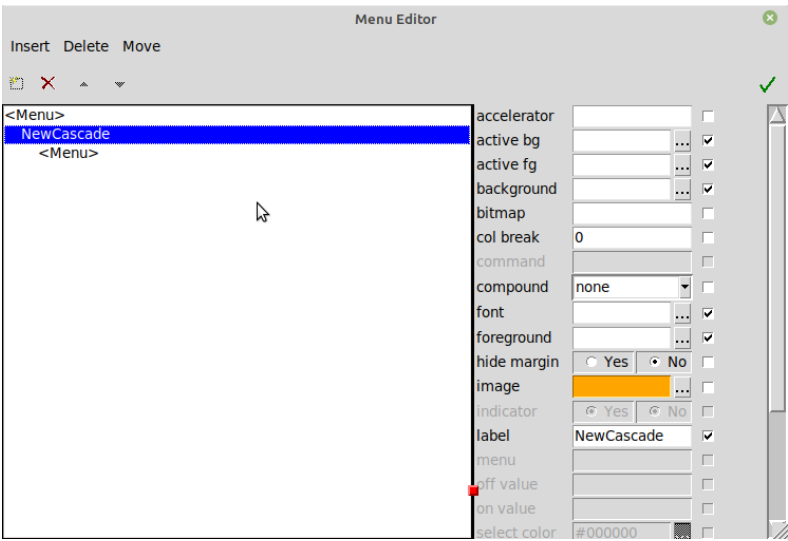
Single Window
Main
Sub 1
Sub 2

Multiple Windows
Main and Sub 1
Main and Sub 2
Sub 1 and Sub 2
All

Exit

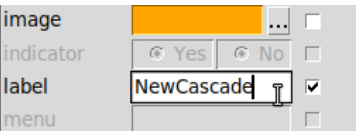
There will be three Cascade items, three separators (represented by “-----”) and each Cascade item will have at least three commands.

So let’s get started designing our Form level Menu. Open the menu editor and select **Insert | New cascade**. Your editor window should look like this.



Be sure to highlight the **NewCascade** item in the menu tree and change the label to **Actions**.

Tip: In order to remove the existing text in the Label box, you have to set the cursor to the end of the label and use the backspace key on the keyboard.



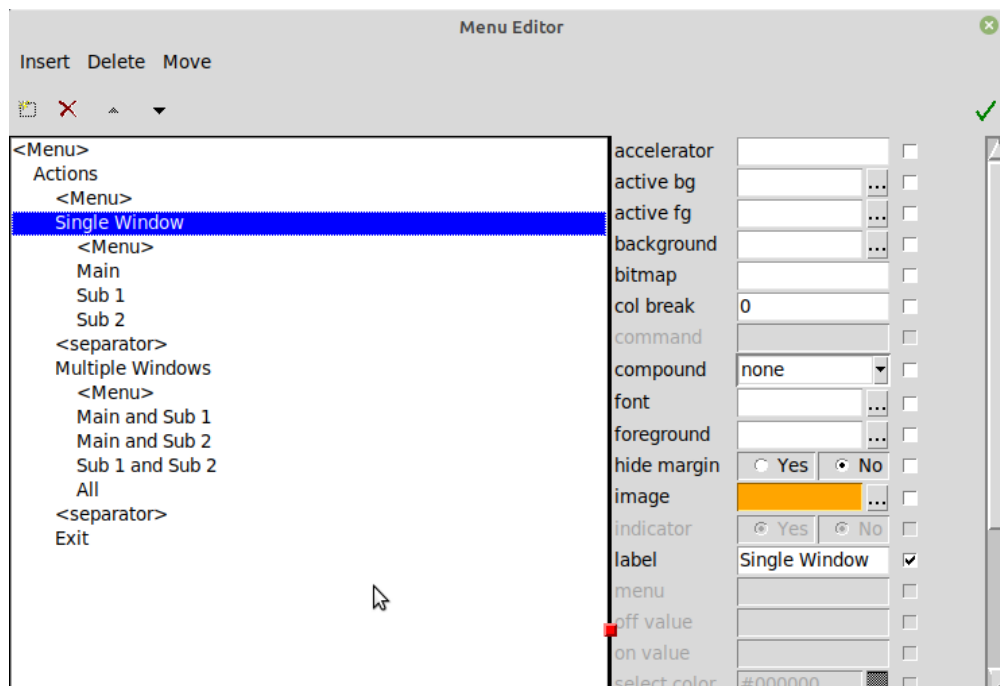
While it is selected, click on **Insert | New Cascade** again and set the label to **Single Window**. Make sure that **Single Window** is selected and click on **Insert | New command** to add our first command. Set the label to **Main** and the set the command attribute to **on_mnuSingleMain** which will set the callback function for this menu item.

I’ve created a grid that contains all of the menu items, it’s parent, label and command (if any). There are other attributes for each menu item, but I’ll leave them for you to discover about on your own after you finish the tutorial. Just be sure to select the parent for any sub items before you insert the next menu item.

Parent	Type	Label	Command
--------	------	-------	---------

	Cascade	Actions	
Actions	Cascade	Single Window	
Single Window	Command	Main	on_mnuSingleMain
SingleWindow	Command	Sub 1	on_mnuSingleSub1
SingleWindow	Command	Sub 2	on_mnuSingleSub2
Actions	Separator		
Actions	Cascade	Multiple Windows	
Multiple Windows	Command	Main and Sub 1	on_mnuMultipleMainSub1
Multiple Windows	Command	Main and Sub 2	on_mnuMultipleMainSub2
Multiple Windows	Command	Sub 1 and Sub 2	on_mnuMultipleSub1Sub2
Multiple Windows	Command	All	on_mnuMultipleAll
Actions	Separator		
Actions	Command	Exit	on_mnuExit

When you've finished, your menu tree should look like this.



When you've finished, click the green arrow on the right side of the editor to close the menu editor. Once you've closed the Menu editor, your main form should look like this. Save your project.



Creating the Button Menu Bar

Now we'll concentrate on creating the Button Menu bar. The first thing we need to do is place a standard Tk **Frame** on our form. It doesn't matter where you put it, since we will set the position manually in the Geometry section of the Attribute Editor.

Once you have placed your Frame, click on the Toplevel form and look at the attribute editor. Near the bottom you will see the Geometry section. Jot down the value in the width section.

Now click back on your Frame to select it and go back to the Attribute Editor.

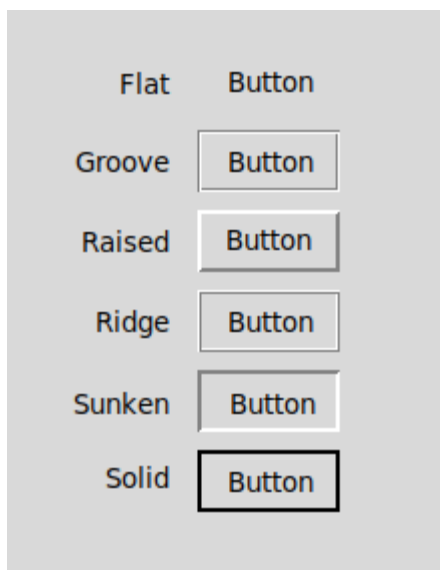
Attribute	Value
Width (Geometry Section)	Width of your Toplevel (Mine is 741)
X position	0
Y position	0
Height	50

So at this point, your form should look like this.



The frame will hold the buttons for our Button menu bar. Now you have a decision to make. Do you want the frame to look flattish or something like the menu bar above it or even something else. While you don't have too many options, you do have a few. You can change the appearance of the Frame by setting the relief attribute.

Your options are:



For my menu bar, I chose to use the Raised relief, which looks pretty much the same as the Menu Bar.

Now we need to gather the graphics for our buttons. I've included a set of graphics with the sample code. The reason we need to do this BEFORE we add our buttons, because the size of the graphics will determine the size of our buttons.

NOTE: I created the images using the free software Inkscape as 80 pixel by 80 pixel images then resized them using another free program called Imagemagik and converted them to a 40x40 pixel image with the command line of

```
convert {source image} -resize 40x40 {output image}
```

We will need to add eight buttons to our Frame. We won't set the images in PAGE, we'll do it in code later on, since it's important to know how to do both.

Since each graphic is 40x40 pixels, we'll make each button 42x42 pixels. You should group them in three buttons near the left side, four buttons near the middle and one button on the right side. That will be our Exit button. For the grouping of the buttons, we'll make each button 50 pixels to the right of the last. Notice that the X position is based on the upper left corner of any widget. The Y position is also based on the upper left corner as well. We'll align all the buttons at a Y position of 3. Here is a graphic that shows the layout of both of our menus in my sample project.



Here are the attributes and values for each button. For the most part, they are the same for each button. I've decided to start each button widget Alias with **"btnBBar"** to group all our button widgets with a common Alias. The same goes for the command attribute for our callbacks. They all start with **"on_btnBBar"**. We'll start from the left and move to the right.

Attribute	Value
Alias	btnBBarShowMain
command	on_btnBBarShowMain
x	10
y	3
width	42
height	42

tooltip text	Show Main Form
---------------------	----------------

Attribute	Value
Alias	btnBBarShow2
command	on_btnBBarShow2
x	60
y	3
width	42
height	42
tooltip text	Show Form 2

Attribute	Value
Alias	btnBBarShow3
command	on_btnBBarShow3
x	110
y	3
width	42
height	42
tooltip text	Show Form 3

Now, we'll work on the four buttons that show the multiple windows. We'll use the same naming scheme. I've started the first button at X position 230 and will maintain a 50 pixel spacing of the buttons.

Attribute	Value
Alias	btnBBarShow1and2
command	on_btnBBarShow1and2
x	230
y	3
width	42

height	42
tooltip text	Show Main and Form 2

Attribute	Value
Alias	btnBBarShow1and3
command	on_btnBBarShow1and3
x	280
y	3
width	42
height	42
tooltip text	Show Form 3

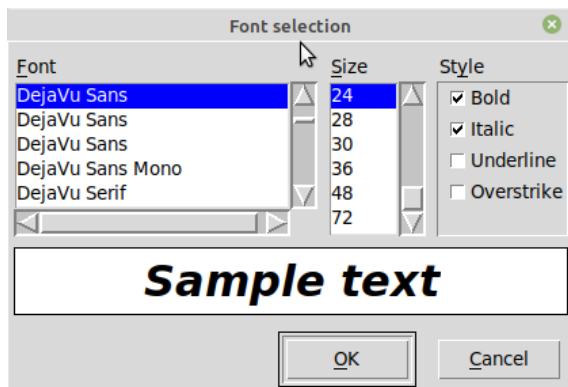
Attribute	Value
Alias	btnBBarShow2and3
command	on_btnBBarShow2and3
x	330
y	3
width	42
height	42
tooltip text	Show Forms 2 and 3

Attribute	Value
Alias	BtnBBarShowAll
command	on_btnBBarShowAll
x	380
y	3
width	42
height	42
tooltip text	Show All Forms

Now we'll finish up with our buttons by doing the Exit button.

Attribute	Value
Alias	btnBBarExit
command	on_btnBBarExit
x	680
y	3
width	42
height	42
tooltip text	Exit the Application

Finally, we'll place a Label in the center of the rest of the form. This is just to make it very obvious that this is the Main form. I used the font dialog (the small button next to the font window with three dots) to set the font to DejaVu Sans 24 point Bold and Italic. If you don't have DejaVuSans as one of your fonts, don't worry, use a Sans Serif font like Arial.



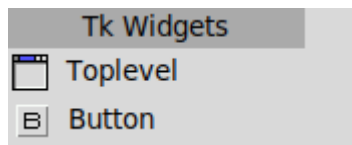
Attribute	Value
Alias	Label1
text	This is the MAIN form
font	-family {DejaVu Sans} -size 24 -weight bold -slant italic
x	146
y	210
width	449
height	111

I used the Multi-Select feature to set the label centered both horizontally and vertically as we did in the previous chapter. The very last thing you want to do is to place your Main form somewhere close to the center of the screen or where ever you want it to appear.

Save your project and generate your Python modules and we'll create our second form. Both the second and third forms will be quick and easy.

Adding our second form

Now we'll add a new Toplevel to our project. Simply click on the Toplevel widget in the Toolkit.



Move it where ever you want it on the screen and we'll set it's attributes. I moved mine to just left and below my main window. You can place it anywhere on the screen. We'll also change the background colour of the form, to make it obvious that it's the second form.

Attribute	Value
Alias	Form2
background	skyblue3
x	503
y	685
width	600
height	450

Next, we'll add a label and a button. The label is similar to the one we did for the Main form. The button will be to hide Form2. To speed things along, I copied the label from the Main form and pasted it into Form 2 and changed it's text and background colour. I then used the MultiSelect feature to centre it both horizontally and vertically.

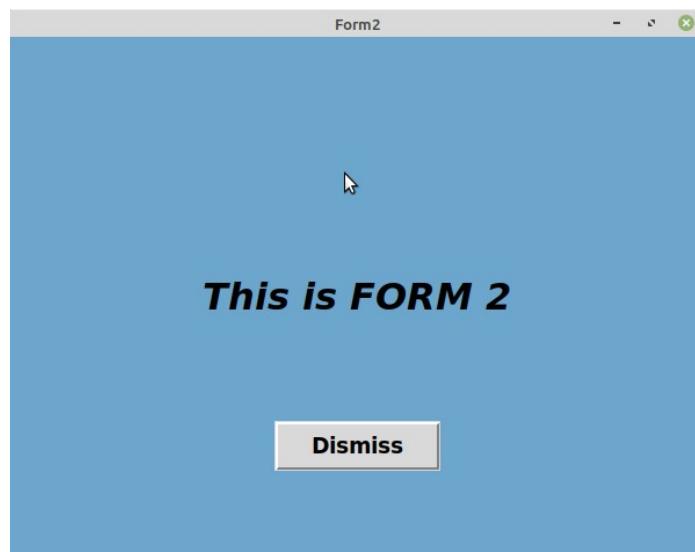
Attribute	Value
Alias	Label1
background	skyblue3
text	This is FORM 2
x	146

y	210
width	449
height	111

For the button, I simply placed one from the toolbox and set it's attributes. I set the font to DejaVu Sans, 14 point bold. I used the MultiSelect feature to centre it horizontally. Notice that I used the alias of btnDismiss2, to avoid naming conflicts with one that we'll place on Form 3.

Attribute	Value
Alias	btnDismiss2
command	on_btnDismiss2
text	Dismiss
x	228
y	331
width	143
height	43
tooltip text	Dismiss This Form

Your second form should look something like this.



Adding the third form

Again, add a new Toplevel and move it to be close to Form2 but to the right of it. I copied the Label and Dismiss button from Form2 to Form3 and set the form alias to Form3 and it's background to **springgreen3**.

Here are the attributes I set for Form3.

Attribute	Value
Alias	Form3
background	springgreen3
x	1118
y	727
width	600
height	450

Next, here are the attributes for the Label for Form3.

Attribute	Value
Alias	Label1
background	springgreen3
text	This is FORM 3
x	146
y	210
width	449
height	111

Finally here are the attributes I set for the Dismiss button for Form3.

Attribute	Value
Alias	btnDismiss3
command	on_btnDismiss3
text	Dismiss

x	228
y	331
width	143
height	43
tooltip text	Dismiss This Form

Your third form should look something like this.



We need to save our file and generate the Python code again, but we need to do something first. You will want to set the positions of your forms, since the form positions are saved in the GUI file. Sometimes, you will want them to be centred in the screen, but sometime you will want them to show up at specific locations. In this case, since we are going to show all three forms at once (on demand), we will set the specific locations before we save the project. PAGE remembers where you have them located when you save and generate your Python files, so use this time to set the form positions.

NOW, save and generate your code.

The Code

We finally get to the interesting part, the code. To me, the code part is always the most fun part. Since PAGE creates so much of the code for us, it's always pretty easy for me and I know that I'm getting close to the end so I can start testing. However, we should define some logic constraints first.

When we show and hide forms, we want to be sure that we always have at least one form on the screen at all times (unless we close the program). We will use a shared module scheme to make sure that all forms can know what is open at any given time. This keeps us from having too many global variables. We need to have a few globals, but this will show you how to use the shared module. The shared module is an empty file named **shared.py** that needs to be located in the project folder and we import it just like any Python library. It's really easy to use, with one thing that you need to remember. Any variables that you want to reference **MUST** be defined before you try to read the value. If you don't you will get an error that the variable does not exist. We'll see more about this when we start working on the **Multiple_support.py** file.

First, let's look at portions of the GUI python module, **Multiple.py**.

Each of our forms is defined as a class. Since we have three forms called Main, Form2 and Form3, there will be a class with each of their names. I'm just going to show you the beginnings of each of the classes.

```
class Main:
    def __init__(self, top=None):
        '''This class configures and populates the toplevel window.
           top is the toplevel containing window.'''
        _bgcolor = '#d9d9d9' # X11 color: 'gray85'
        _fgcolor = '#000000' # X11 color: 'black'
        _compcolor = '#d9d9d9' # X11 color: 'gray85'
        _ana1color = '#d9d9d9' # X11 color: 'gray85'
        _ana2color = '#ecec' # Closest X11 color: 'gray92'

        top.geometry("741x532+743+132")
        top.minsize(1, 1)
        top.maxsize(4225, 1410)
        top.resizable(0, 0)
        top.title("Main Window")
        top.configure(highlightcolor="black")
        ...
```

Notice that all of the classes have a line that defines the geometry of the form. This provides the width, height, x position and y position of the form on the screen. Here is the geometry line for the Main form...

```
top.geometry("741x532+743+132")
```

Scroll down in the file and you will find the class for Form2. You will see it is very similar to our Main form, but shorter, since it does not have as many widgets as the Main form does. This is just the top portion of the class definition.

```
class Form2:
    def __init__(self, top=None):
        '''This class configures and populates the toplevel window.
            top is the toplevel containing window.'''
        _bgcolor = '#d9d9d9' # X11 color: 'gray85'
        _fgcolor = '#000000' # X11 color: 'black'
        _compcolor = '#d9d9d9' # X11 color: 'gray85'
        _ana1color = '#d9d9d9' # X11 color: 'gray85'
        _ana2color = '#ecec' # Closest X11 color: 'gray92'

        top.geometry("600x450+496+728")
        top.minsize(1, 1)
        top.maxsize(4225, 1410)
        top.resizable(0, 0)
        top.title("Form2")
        top.configure(background="skyblue3")
        ...
```

Continue scrolling down and you will find the class definition for the third form. It should be about the same size as the class for Form2, since it contains the same number of widgets.

```
class Form3:
    def __init__(self, top=None):
        '''This class configures and populates the toplevel window.
            top is the toplevel containing window.'''
        _bgcolor = '#d9d9d9' # X11 color: 'gray85'
        _fgcolor = '#000000' # X11 color: 'black'
        _compcolor = '#d9d9d9' # X11 color: 'gray85'
        _ana1color = '#d9d9d9' # X11 color: 'gray85'
        _ana2color = '#ecec' # Closest X11 color: 'gray92'
```

```
top.geometry("600x450+1118+727")
top.minsize(1, 1)
top.maxsize(4225, 1410)
top.resizable(0, 0)
top.title("Toplevel 2")
top.configure(background="springgreen2")
...
```

Go ahead and close the Multiple.py file. At this point, we need to start modifying the Multiple_support.py file. Open it up in your editor or IDE. As always, we'll start with the import section. As I always do, I'll set any lines you need to modify or add to a bold font.

```
import sys
import shared # empty file shared.py in the project folder
import tkinter as tk
import tkinter.ttk as ttk
from tkinter.constants import *

import Multiple
```

Since we only need to import the shared.py file there is only one line that needs to be added. (You can leave off the comment if you really want to, but it might help you to remember if you ever come back to the project what the shared.py file is.)

Here is the **main** function that PAGE creates for us. Remember from our previous chapters, this is where everything gets started and where we will create the call to our setup function that will be run just before the user sees our form(s).

```
def main(*args):
    '''Main entry point for the application.'''
    global root
    root = tk.Tk()
    root.protocol('WM_DELETE_WINDOW', root.destroy)
    # Creates a toplevel widget.
    global _top1, _w1
    _top1 = root
```

```
_w1 = Multiple.Main(_top1)
# Creates a toplevel widget.
global _top2, _w2
_top2 = tk.Toplevel(root)
_w2 = Multiple.Form2(_top2)
# Creates a toplevel widget.
global _top3, _w3
_top3 = tk.Toplevel(root)
_w3 = Multiple.Form3(_top3)
# Process the startup function
startup()
root.mainloop()
```

One thing that I want to point out here, is that each of our forms is actually set up here. When we only had one form, it was named **_top1** and we referenced it as **_w1**. Now that we have three forms, there is **_top1**, **_top2**, **_top3**, **_w1**, **_w2**, **_w3**. Just as before we use **_wn** to refer to a widget on the form, where **n** is the form number. So **_w3** would refer to a widget on form #3.

At this point, we need to create the `startup()` function.

```
def startup():
    # Set the shared module variables to False
    # If we don't we will get an error that the variable doesn't exist
    shared.MainActive = True
    shared.Top2Active = False
    shared.Top3Active = False
    # Hide forms 2 and 3
    hide_me_2()
    hide_me_3()
    # Load the button images, since we didn't do it in PAGE
    load_btn_images()
```

The first thing we do (so we don't forget them) is to set up our shared variables. The reason we do it here, and at the very top of our startup function, is that we will be using them in the `hide_me` functions. We start the `MainActive` in a `True` state, since this will be the only visible form when we start the program. We set `Top2Active` and `Top3Active` a `False` state, because we are going to hide the forms when we start the program. We finally call a function called `load_btn_images`, which like the name suggests, loads all the buttons in our button bar menu to their appropriate images.

As I state in the comment at the top of the function, when a Tkinter widget has an image, that image must be associated with a global variable. This is because of Python's garbage collection processes. Python doesn't know that you want that image to stay showing on the widget the entire time the program runs. So when the garbage collection tries to free memory, if there is no global associated with the variable, the image will go away. Once we have the globals defined, then we assign each image as an object to the proper variable using the `tk.PhotoImage` method, passing the proper path and filename of that image, then we use the `configure` method of the button (in this case) to set the image attribute to our object variable. We do this for each and every button. I put comments stating the code associated with each button, but removed them here just to keep things a bit shorter. They still are in the code provided with the tutorial.

```
def load_btn_images():
    # Images need a global variable so that Python's garbage collection doesn't
    #   clear the image accidentally.
    global img1, img2, img3, img4, img5, img6, img7, img8
    # Image for button 1
    img1 = tk.PhotoImage(file="./images/img1-40.png")
    _w1.btnBBarShowMain.configure(image=img1)
    # Image for button 2
    img2 = tk.PhotoImage(file="./images/img2-40.png")
    _w1.btnBBarShow2.configure(image=img2)
    img3 = tk.PhotoImage(file="./images/img3-40.png")
    _w1.btnBBarShow3.configure(image=img3)
    img4 = tk.PhotoImage(file="./images/img-Multi-1-2-40.png")
    _w1.btnBBarShow1and2.configure(image=img4)
    img5 = tk.PhotoImage(file="./images/img-Multi-1-3-40.png")
    _w1.btnBBarShow1and3.configure(image=img5)
    img6 = tk.PhotoImage(file="./images/img-Multi-2-3-40.png")
    _w1.btnBBarShow2and3.configure(image=img6)
    img7 = tk.PhotoImage(file="./images/img-Multi-all-40.png")
    _w1.btnBBarShowAll.configure(image=img7)
    img8 = tk.PhotoImage(file="./images/system-shutdown.png")
    _w1.btnBBarExit.configure(image=img8)
```

I usually keep my most of my support functions near the bottom of the support module. I do this, so I can find them easier. Usually, the only functions I keep near the top are the startup function and maybe one or two functions that get called early on, like the `load_images` function. We'll fast forward to the show and hide functions. That way, if you want to test your code early, any functions we need from our startup code will be written and available. Since PAGE creates all of our callback functions as skeletons, when they get called, there is enough code there to allow for testing thanks to the debugging print statements.

```
# =====  
# Show and hide functions  
# =====  
  
def show_me_Main():  
    global _top1  
    _top1.deiconify()  
    shared.MainActive = True  
  
def hide_me_Main():  
    global _top1  
    _top1.withdraw()  
    shared.MainActive = False
```

Let's deal with the show and hide functions of the Main form. The others are pretty much the same thing with only the variable pointers changed as needed.

The **hide_me_Main** function will hide the form by using the Tkinter **root.withdraw** method of the actual form. This forces the form to be **iconized** or sent to an unseen state. It's still there, but it's hidden from view. The **show_me_Main** function restores the form to it's last position. We also use one of our shared variables, **MainActive**, setting it to **True** if the Main form is being shown, or to **False** if it is minimized.

As I just stated, the `show_me` and `hide_me` functions for forms two and three are pretty much the same. We use **_top2** to hide and show form2 (rather than **root**) and we set the shared variable **Top2Active** to **True** or **False** depending on the state of the form. The **show_me_3** and **hide_me_3** functions do the same thing just with the third form and it's shared variable.

```
def show_me_2():  
    global _top2
```



```
_top2.deiconify()
shared.Top2Active = True

def hide_me_2():
    global _top2
    _top2.withdraw()
    shared.Top2Active = False

def show_me_3():
    global _top3
    _top3.deiconify()
    shared.Top3Active = True

def hide_me_3():
    global _top3
    _top3.withdraw()
    shared.Top3Active = False

# =====
# End of Show and hide functions
# =====
```

Now we are at the point that we can start filling in the skeleton callback functions that PAGE created for us. The first two are the **on_mnuExit** function and the **on_btnBBarExit** function. You can see all we really need to do in these are to simply add the line **root.destroy()**, which will politely terminate the program.

```
def on_mnuExit(*args):
    print('Multiple_support.on_mnuExit')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    root.destroy()
```

```
def on_btnBBarExit(*args):
    print('Multiple_support.on_btnBBarExit')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    root.destroy()
```

Remember, if you don't want or need to see the debugging code for these functions, feel free to comment them out or delete them. PAGE puts them in so that when you are starting your coding process and debugging so you know that the event has been captured and responded to.

Remember, earlier I laid out some logic constraints for the project. The most important one is that we will always have at least one form on the screen unless we are closing the program. That's why we created the three shared variables MainActive, Top2Active and Top3Active. When we need to hide a form, we can use an if statement to make sure that it isn't displayed. We'll start with the form menu callback function on_mnuMultipleAll.

```
def on_mnuMultipleAll(*args):
    print('Multiple_support.on_mnuMultipleAll')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if not shared.Top2Active:
        show_me_2()
    if not shared.Top3Active:
        show_me_3()
    if not shared.MainActive:
        show_me_Main()
```

When the user clicks on this menu item, we want to display all three forms. We could just call the three show_me functions here, without any additional logic. However, good programming practices will say that if a form is already visible on the screen, we don't want to call show_me function again. Therefore, we will check to see if the shared variable for that form is set to True before calling the show_me function for that form.

Using the same logic, when we want to show just the Main form and Form2, we can use the logic below. This makes sure that Form3 will no be shown when this callback runs.

```
def on_mnuMultipleMainSub1(*args):
    print('Multiple_support.on_mnuMultipleMainSub1')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if not shared.Top2Active:
        show_me_2()
    if shared.Top3Active:
        hide_me_3()
    if not shared.MainActive:
        show_me_Main()
```

Again, when we want to show the Main form and Form3, but not Form2, we can use this logic set.

```
def on_mnuMultipleMainSub2(*args):
    print('Multiple_support.on_mnuMultipleMainSub2')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.Top2Active:
        hide_me_2()
    if not shared.Top3Active:
        show_me_3()
    if not shared.MainActive:
        show_me_Main()
```

Here are the callback functions for the rest of the form menu callbacks that we defined.

```
def on_mnuMultipleSub1Sub2(*args):
    print('Multiple_support.on_mnuMultipleSub1Sub2')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if not shared.Top2Active:
        show_me_2()
    if not shared.Top3Active:
```

```
        show_me_3()
    if shared.MainActive:
        hide_me_Main()

def on_mnuSingleMain(*args):
    print('Multiple_support.on_mnuSingleMain')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.Top2Active:
        hide_me_2()
    if shared.Top3Active:
        hide_me_3()
    if not shared.MainActive:
        show_me_Main

def on_mnuSingleSub1(*args):
    print('Multiple_support.on_mnuSingleSub1')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if not shared.Top2Active:
        show_me_2()
    if shared.Top3Active:
        hide_me_3()
    if shared.MainActive:
        hide_me_Main()

def on_mnuSingleSub2(*args):
    print('Multiple_support.on_mnuSingleSub2')
    for arg in args:
        print('another arg:', arg)
```

```
sys.stdout.flush()
if shared.Top2Active:
    hide_me_2()
if not shared.Top3Active:
    show_me_3()
if shared.MainActive:
    hide_me_Main()
```

The logic for the button bar callbacks is exactly the same as the form menu callbacks that show and hide the various forms.

```
def on_btnBBarShowMain(*args):
    print('Multiple_support.on_btnBBarShowMain')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.Top2Active:
        hide_me_2()
    if shared.Top3Active:
        hide_me_3()
```

```
def on_btnBBarShowland2(*args):
    print('Multiple_support.on_btnBBarShowland2')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.Top3Active:
        hide_me_3()
    show_me_Main()
    show_me_2()
```

```
def on_btnBBarShowland3(*args):
    print('Multiple_support.on_btnBBarShowland3')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.Top2Active:
        hide_me_2()
    show_me_Main()
    show_me_3()
```

```
def on_btnBBarShow2(*args):
    print('Multiple_support.on_btnBBarShow2')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.MainActive:
        hide_me_Main()
    if shared.Top3Active:
        hide_me_3()
    show_me_2()
```

```
def on_btnBBarShow2and3(*args):
    print('Multiple_support.on_btnBBarShow2and3')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if shared.MainActive:
        hide_me_Main()
    show_me_2()
    show_me_3()
```

```
def on_btnBBarShow3(*args):
```

```
print('Multiple_support.on_btnBBarShow3')
for arg in args:
    print('another arg:', arg)
sys.stdout.flush()
if shared.MainActive:
    hide_me_Main()
if shared.Top2Active:
    hide_me_2()
show_me_3()
```

```
def on_btnBBarShowAll(*args):
    print('Multiple_support.on_btnBBarShowAll')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    show_me_Main()
    show_me_2()
    show_me_3()
```

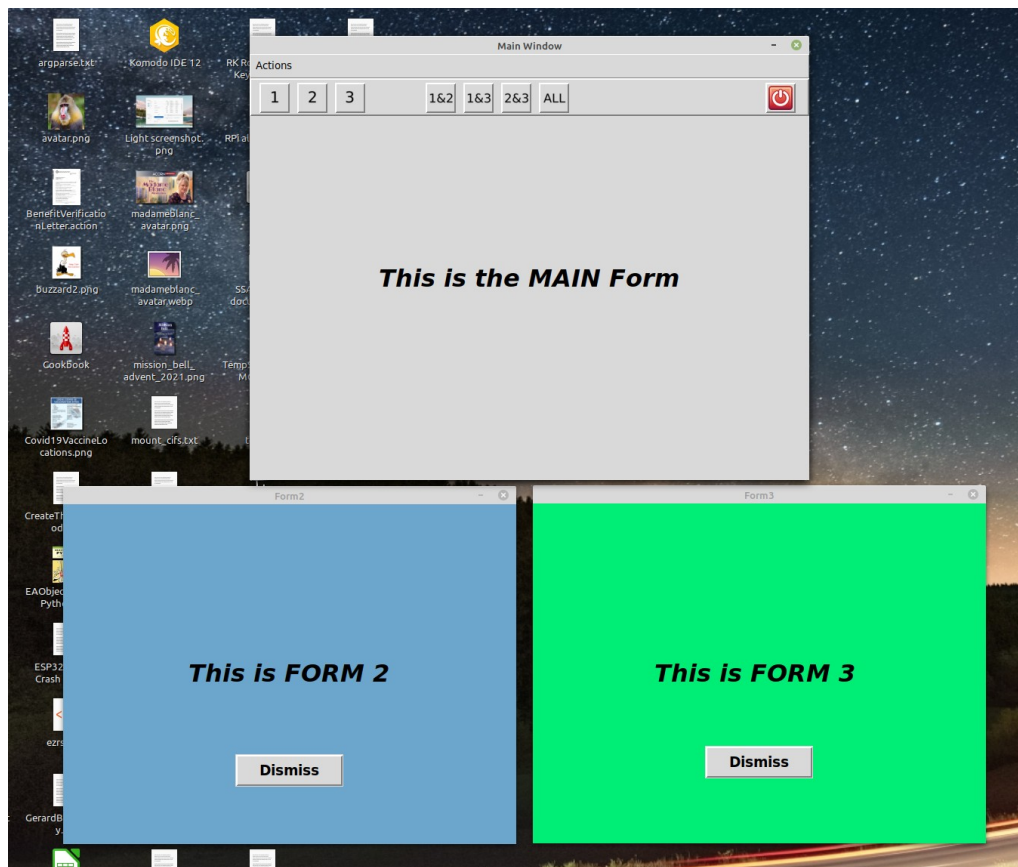
The astute programmers among you would say, “why duplicate the code for each of the callbacks?” Most programmers would simply make a single callback and point to it from both the form menu and the button menu. However, I wanted to make sure you fully understood the logic.

Finally, we have the dismiss form callbacks for Form2 and Form3. Since we want to make sure that at least one form is shown at all times and we are closing one of the sub forms (Form2 or Form3), we need to make sure that when we dismiss one of those forms, we hide it and then show the Main form.

```
def on_btnDismiss2(*args):
    print('Multiple_support.on_btnDismiss2')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    if not shared.MainActive:
        show_me_Main()
    hide_me_2()
```

```
def on_btnDismiss3(*args):  
    print('Multiple_support.on_btnDismiss3')  
    for arg in args:  
        print('another arg:', arg)  
    sys.stdout.flush()  
    if not shared.MainActive:  
        show_me_Main()  
    hide_me_3()
```

That's it for the code. When you run your program and select one of the Show All functions, your program should look like this...



While the second and third forms are really simple, you can see that what you've already learned from our previous projects in this tutorial will make it easy to create complicated applications with multiple forms.

Conclusion

I truly hope that you've enjoyed going through the tutorial.

We've set up a new support forum on Discord to assist anyone having issues with PAGE. The invite link for the forum is <https://discord.gg/V6zCcjq> and will never expire. If you have never used Discord, you can get many tutorials on installing and using the apps. There are versions that work for most any web browser as well as applications for Android, Windows, Linux and Mac and I'm sure IOS. Feel free to come to Discord and share your questions as well as your tips and applications.

Greg