

Learning Page - A Python GUI Designer



By G.D. Walters

Copyright © 2018 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.....	7
Requirements.....	8
Windows.....	8
Linux.....	8
OSX.....	8
Installation.....	8
Windows.....	8
Linux.....	9
OSX.....	9
Chapter 1 - Getting Started.....	11
Our First Program.....	14
The Main Window - Toplevel.....	14
Adding a Button.....	16
Chapter 2 - Moving on.....	21
Buttons.....	21
Frames.....	21
Labels.....	22
Single line entry text boxes.....	22
Check Buttons.....	22
Radio Buttons.....	22
Beginning the Project - Layout.....	22
Step 2 - Bindings.....	27
Step 3 – The Code.....	28
Chapter 3 – A more realistic project.....	31
Building the UI.....	32
The Code.....	35
Chapter 4 – Linking A Media Player To Searcher.....	43
Building the UI.....	44
Creating the Menu Bar.....	44
The Code.....	46
The About Box.....	51
The About Box Form.....	51
The About Box Code.....	52
Linking The About Box To The Media Player.....	54
Linking The Media Player To The Searcher Program.....	55

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.

Now, I find myself going back to some of my roots and working on some Python projects that require a GUI. I was talking with my son last night about Page and its capability for a project he wants to work on and decided to look it up and see what the latest version was. To my delight, it was up to version 4.9 and will soon be version 4.11 to be released in early 2018.

I downloaded it to my laptop and found out that the tutorials I previously wrote are no longer relevant (in some of the specific points) due to changes in the Page program, so I decided to update the tutorials and maybe come up with some more relevant examples to help people along their discover/rediscover journey of Page.

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 for this document. 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 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.

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

Windows

Under Windows, you will need, of course, Python, either 2.7x or 3.x. I will be using Python 2.7.9.

You should also download ActiveTcl 8.6.6 (or the latest version you can get) from ActiveState. I am using the free community version.

You will also want to have some sort of an editor to keep up with your Python code. I use a very nice free cross-platform IDE (Integrated Development Environment) called Geany.

Linux

While Linux usually has everything you need, you will probably want to use ActiveTcl version 8.6.6 from ActiveState. I use the free community version.

I also use (as with Windows) a very nice free IDE called Geany.

OSX

Since I don't have access to any OSX machines, I really can't speak to this from direct experience, but I understand from Don that the requirements for OSX are the same as Linux.

Installation

Windows

Download the latest version of ActiveTCL from <https://www.activestate.com/activetcl/downloads> and run the file.

Many Windows users have a problem when they start Page. It complains that the program wish.exe can not be found. This is because tcl hasn't been installed.

Download the latest version of Page from <https://sourceforge.net/projects/page/> and run the file. This will install Page.

Finally, create a “master” directory on your hard drive to hold your files.

Linux

Installation under Linux is a bit more time consuming than Windows, but not terrible.

First you will want to download ActiveTcl from ActiveState at <https://www.activestate.com/activetcl>. Again, I'm using the free community version. The download is a tar-gzipped archive. Once the file is downloaded you should follow the following steps for installation.

- In a terminal, type **tar xzf /path/to/ActiveTcl-download.tar.gz**.
- Change to the folder you extracted the files to and run the installer script. You should do this as root, since the installation goes into the opt folder. **sudo ./install.sh**
- Ensure that your PATH variable includes the directory that contains the installed executable files. **export PATH="/opt/ActiveTcl-8.6/bin:\$PATH"**

That's it for installing ActiveTcl. Now download the Page distribution

<https://sourceforge.net/projects/page/> . The page normally detects your OS, so it should prompt you to download the tar-gzipped latest version.

Untar the downloaded file into your home directory.

Run “**./configure**” in the installation directory. This generates a script that invokes Page.

Remove any “.pagerc” files in the folder.

OSX

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

I'll leave the installation of Geany or whatever IDE you choose to use to you.

As under Windows, create a “master directory” to hold your development files.

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). Again, your choice of an IDE is YOUR choice. I like Geany, others like Sublime Text while others like IDLE.

Chapter 1 - Getting Started

To start Page, click on the icon. After a few seconds, you should be presented with three (at least) new windows on your desktop workspace. It should look something like this...

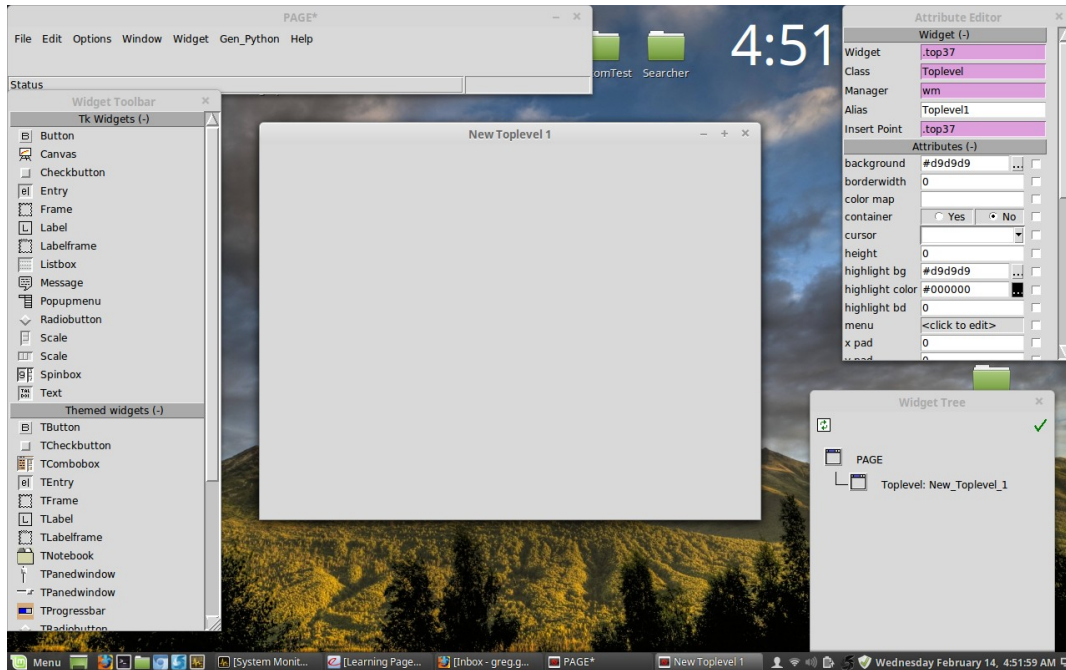


Image 1: The 5 Main Page Windows

Clockwise from the top left, they are the main window, the Attribute Editor, Widget Tree and the Widget Toolbar. The Toplevel widget is added when you start Page (as of version 4.10). Below, you can see them each.

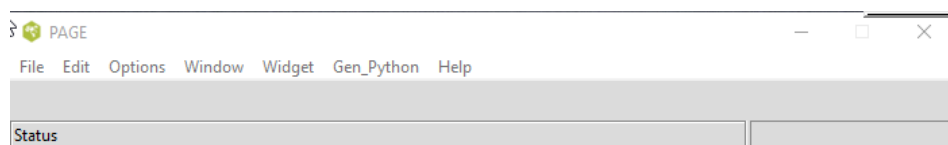


Image 2: The Page Main Window

The main window holds the standard File (Open | Save, etc.) as well as options to deal with the various windows, Python generation and more. Minimizing this window will minimize all other Page windows as well, except your Toplevel widget.

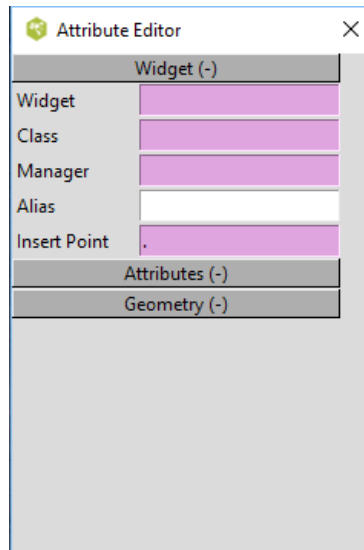


Image 3: The Attribute Editor

The Attribute Editor window will be one of the most used sections during your GUI creation sessions. Here you will set up the important visual parts of the various widgets in your project like position, size, color, text, alias (the name of the widget in code) and more.

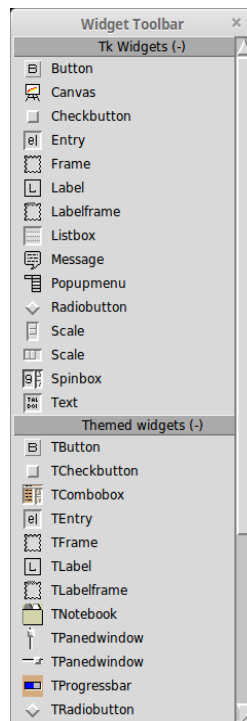


Image 4: The Widget Toolbar

The toolbox holds all of the widgets that you will use to design your GUI. These widgets include things like static text objects like labels, text entry objects, buttons including check and radio buttons, combo boxes and much much more.

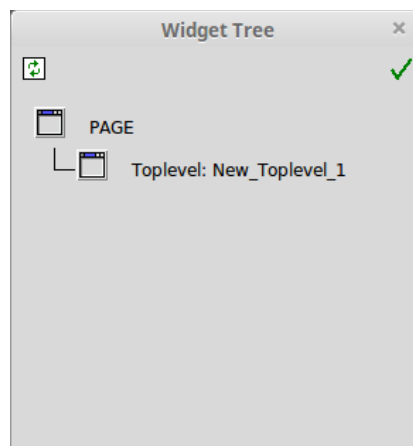


Image 5: The Widget Tree

The Widget Tree shows you all your widgets in a hierarchical view. You can click on any of the widgets there to select them in the main designer, or to change the attributes. You can also right click on a widget here to select other options like bindings.

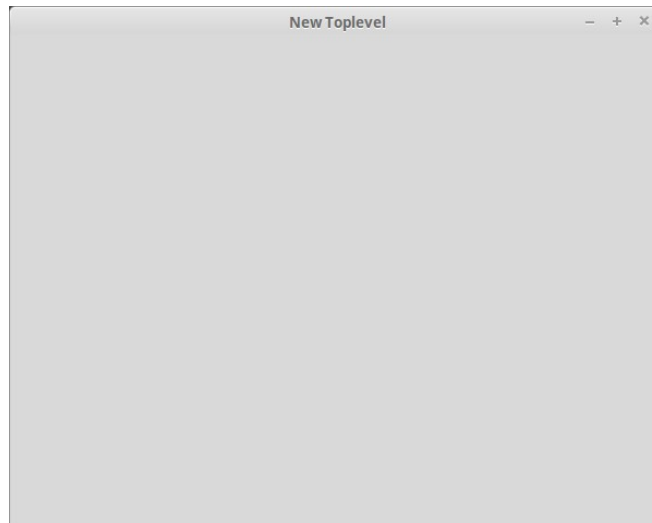


Image 6: New Toplevel Widget

Finally, as of Page 4.10, A new Toplevel widget will automatically be presented for you when you start Page with no command line parameter. If you want to rework an older project, simply start Page with the Project .tcl file (along with the path).

Our First Program

Our first program will simply include a GUI window with a single button that closes and destroys the window.

The Main Window - Toplevel

To start a new project, simply start up Page. This will open up a blank form (along with the other windows you will need) that will be the main form for your application. It will look something like this...



Image 7: The Toplevel Widget On Startup

Consider this the blank canvas that will hold your masterpiece. It can be moved around the screen and/or resized to fit your purposes. Once you have placed it where you want the main window to appear when you start your app, you need to change the title. By default, it is titled “New Toplevel”. Let’s change it to “My First Page App”. To do this, you will use the Attribute Editor. Use the scroll bar on the right of the Attribute Editor, scroll down to the line ‘**title**’ which is just above the Geometry bar. Change the text that is there from “New Toplevel” to “My First Page App”. By pressing the [Enter] key on the keyboard, it will “set” the change. Now scroll back up to the top of the Attribute Editor window. There you will see five options under the ‘Widget’ section. The only one you should change in this section is the one marked ‘**Alias**’. This is what we will call the form/widget in our program to easily reference it. Make sure you use something descriptive, but in only one word. For our learning purposes, we’ll call it “MainForm”. Enter that into the text box next to the word “Alias”.

You might notice that a new window has popped up that is named “Widget Tree”. This will show a hierarchical representation of your form and all the widgets associated to it. For now, you can just click the checkmark to dismiss it, or you can move it somewhere out of the way. If you ever want to see it again and can’t find it, you can get to it from the **main window** | **Window** | **Widget Tree** or simply by pressing <Alt>W.

Before going any further, you should save your work. You’ll want to do this often, especially while you are learning. On the Main Window, select **File** | **Save**. You will be prompted to select the folder and name of the project. For now, use “FirstApp” as a filename. You might also want to put it into a

separate folder to organize your files. Also notice, that this file is going to be saved with a “.tcl” extension. That’s the way it should be, since we are creating the Tcl script for the GUI. There won’t be any Python code for a while. This file will allow you to rework the GUI at a later time.

Adding a Button

Now, let’s put a simple button somewhere near the middle of our form that will exit the form.

On the Widget Toolbar, you will see “**Button**” at the top of the toolbox. Click on “**Button**”, then click somewhere in our main form.

The button we just created will have eight black squares surrounding it. These are handles that you can use to resize the widget. You can also click and drag the widget to where ever you want it to reside within the form.

Once we have place the button widget, go to the Attribute Editor and set the alias to “**btnExit**”. Scroll down until you find the **text** attribute and change it to “**Exit**”. As soon as you press enter, you will see the label on the button in our form change from “Button” to “Exit”. Now use the handles on the button widget to make it a bit wider. Once you have it the way you like it, save your .tcl file again to keep it up to date.

We have a number of things to do before we are done, but we’ll do that in a moment. For now, we can generate our Python code to see how it all works.

On the main Page window menu, you will 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 it to see what the code looks like, but don’t change anything. Next, press the **Save** button, which will save the Python code that Page has generated for us.

You will notice that there is a **Run** button on this window. If you click it now, you will get a message box notifying you that an error has occurred with the message “*No support module has been created and saved.*” That’s ok, since that’s what we’ll do next. Click the **Ok** button on the message box, and then click on the **Close** button on the Generated Python window.

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 and when you are done click **Save**. Now click the **Run** button. In a few seconds, you will see your First App window. It should look something like this...

Learning Page - A Python GUI Designer

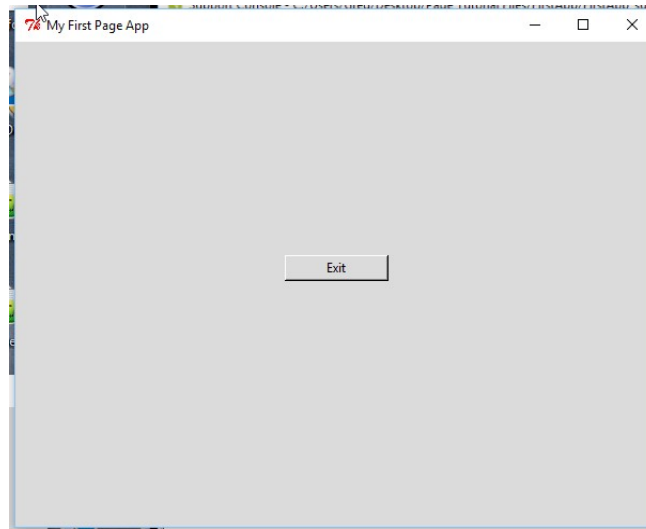


Image 8: Our First Page App


Feel free to click on the **Exit** button, but nothing will happen yet, because we haven't written any code to handle the mouse click. That's ok. We'll do that in a moment. For now, close the demo window (the "X" in the upper right of the window) and close the code editor.

You might be wondering why there is a need for a support module that is separate from the main file. This is done so that the main Python file (in this case "FirstApp.py") is kept to only the definitions needed to create the UI. All other functions, like the code that controls what happens when you click on the exit button, are kept in the support module (in this case named "FirstApp_support.py"). This makes it easy to maintain your code without accidentally changing the UI definition file and you can tweak or modify the UI without having to copy and paste your code from a safety copy. By the way, every time you save your files in Page, it makes a backup of the previous file, so you can always go back to see what you had.

Tip: Typically, you will only want to generate the support module when you are done with your GUI design. There is a chance that the code you have put into the support module might get scrambled by the new code that Page puts in. The errors will be small and usually simply require cut and paste of a few lines of code to rearrange things. Don has done a very good job of making sure that there aren't any big stumbling blocks here.

In order to get the Exit button to work, we need to create what is called a **binding**. This creates code that connects the widget, in this case the Exit button, to a function that will call the `sys.exit()` routine to exit and destroy this window.

To create the binding, right click our `btnExit` in the designer window and select **Bindings...** from the popup menu. This will open another window that makes it easy to create various types of relations

between widget events and our code. In this case, we will want to make sure that **btnExit** is highlighted then press the **Add** button (far left button ) on the tool bar. You will see a menu pop up that looks something like the image below.

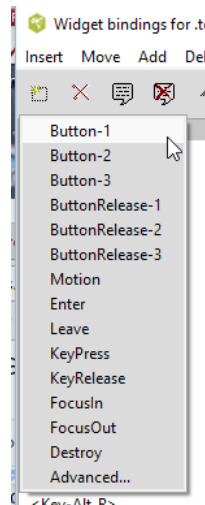


Image 9: Binding Options

There are a number of options here, and we are going to select **Button-1**. This equates to the left mouse button. Button-2 is the middle button (if your mouse has a middle button) and Button-3 is the right button. There are a number of other options, some of which bind other events like those that fire on the button release, key presses and more. We'll explore some of these later on in this document.

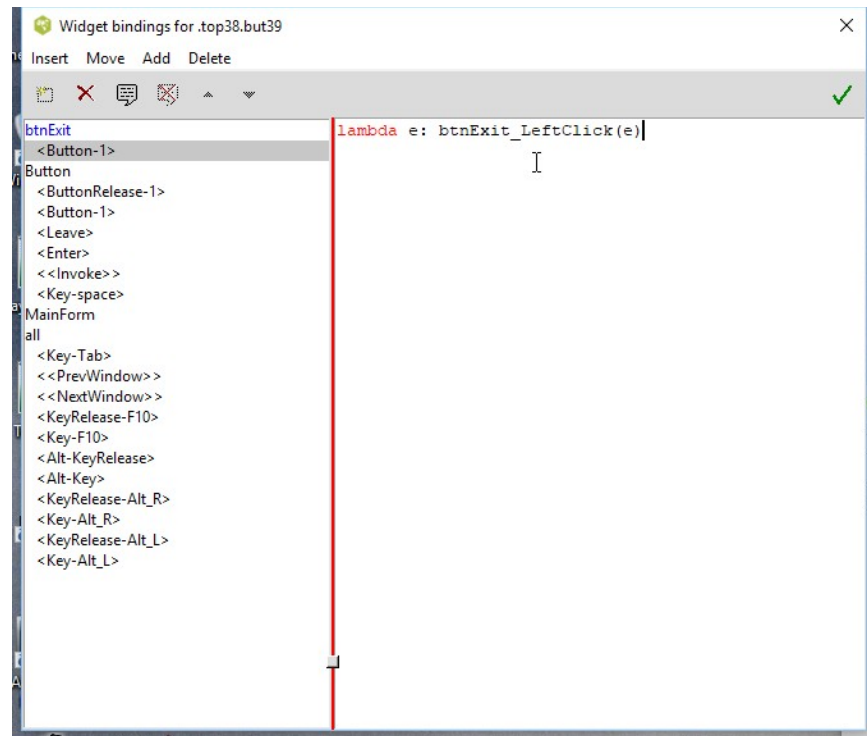


Image 10: The Binding Window

Once you select Button-1, you should see “lambda e: xxx(e)” appear in the right side panel. This is our binding line. We need, however, to change this somewhat to make it a bit more self-documenting. We’ll change the “xxx(e)” to “**btnExit_LeftClick(e)**”. Highlight the “xxx” and replace it with “**btnExit_LeftClick**”.

Now, save the project, Generate the Python code and then the support file. When you generate the support file, you will get a message box that tells you that the support file already exists, along with three options. One is to replace the file, one to use the existing file and one to update the file. At this point, we really haven’t done anything in the existing file, so it wouldn’t be a great loss if we replaced it with a new version. However, once you get doing more, replacing the file will cause you to lose any changes or custom code you have already entered. So to be safe, select Update.

Now, if you compare the original FirstApp.py to the new one that was just generated, you will see that an additional line has been generated that contains our binding at the bottom of the button definition. It is...

```
self.btnExit.bind('<Button-1>',lambda e:FirstApp_support.btnExit_LeftClick(e))
```

Tip: You might notice that in the above code line, the parameter that is passed to the `_LeftClick` event handler routine is called (e). In the code generated by Page below in the support file, it is called (p1). That's because the binding of an event passes information about the event that includes a large

amount of information to the handler function. If you don't need to pass any parameters, you can use the command attribute of the button or other widget.

The support file also has the new routine `btnExit_LeftClick(p1)` already generated for us.

```
def btnExit_LeftClick(p1):
    print('FirstApp_support.btnExit_LeftClick')
    sys.stdout.flush()
```

If you don't add any more code, when you run the app now, the terminal will show "FirstApp_support.btnExit_LeftClick" each time you left click the Exit button.

Now to make the button actually function the way we want it to, which is to close this window, we need to add the following to the `btnExit_LeftClick` routine in the support file. Page provides a function called `destroy_window()`, which is the proper way to end our GUI program. We'll call that code from here.

```
destroy_window()
```

Now the routine should look like this.

```
def btnExit_LeftClick(p1):
    print('FirstApp_support.btnExit_LeftClick')
    sys.stdout.flush()
    destroy_window()
```

Now when you run the program, clicking the Exit button will cause the window to close.

Now you can see that it is pretty easy to create GUI programs for Python using Page. Next we will create a more complicated program demonstrating some of the various widgets available to us within Page.

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.
- Radio Buttons: Demonstrates the grouping function of the radio buttons and setting a label with text dynamically.
- Check Buttons: 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. 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 One-Of-Many widget set. Only one Radio button may be selected (On) state within a group at any time and are 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...

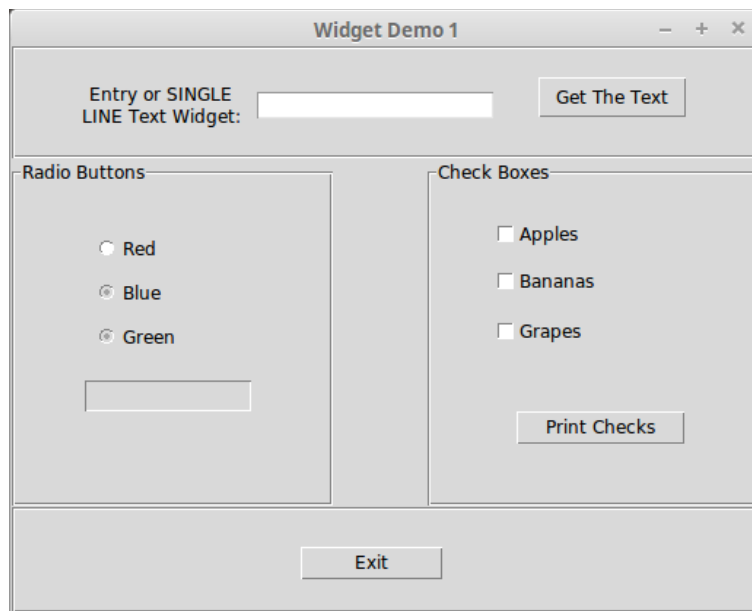


Image 11: Second App

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.

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:

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 popup 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:

alias: **Label1**
 text: **Entry or SINGLE LINE Text Widget:**
 width: **116** (This is different from the width in the Geometry section)
 wrap length: **110**
 x position: **30**
 y position: **20**
 width: **146**
 height: **41**

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:

alias: **entryExample**
 x position: **167**
 y position: **30**
 width: **164**
 height: **20**

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

alias: **btnGetText**
 text: **Get The Text**
 x position: **360**
 y position: **30**

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 Radio Buttons. We'll start by placing a Labelframe widget near the middle of the Toplevel window. Once that is done, set its attributes as follows:

alias: **IframeRadioButtons**
text: **Radio Buttons**
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.

alias: **rbRed**
text: **Red**
value: **Red**
variable: **Colors**
x position: **50**
y position: **50**

alias: **rbBlue**
text: **Blue**
value: **Blue**
variable: **Colors**
x position: **50**
y position: **80**

alias: **rbGreen**
text: **Green**
value: **Green**
variable: **Colors**
x position: **50**

y position: **110**

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 is somewhere in the left frame that is empty.

alias: **lblRbInfo**
 relief: **sunken**
 text variable: **Colors**
 x position: **50**
 y position: **150**
 width: **114**
 height: **21**

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

alias: **lframeCheckBoxes**
 text: **Check Boxes**
 x position: **286**
 y position: **77**
 width: **230**
 height: **235**

Now add three check buttons and set the attributes as follows.

alias: **chkApples**
 text: **Apples**
 variable: **che39**
 x position: **40**
 y position: **40**

Check Button #2

alias: **chkBananas**
 text: **Bananas**
 variable: **che40**
 x position: **40**
 y position: **72**

Check Button #3

alias: **chkGrapes**
 text: **Grapes**


```
variable: che41  
x position: 40  
y position: 106
```

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

```
alias: btnPrintChecks  
text: Print Checks  
x position: 60  
y position: 170  
width: 117  
height: 24
```

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 app.

```
alias: frameBottom  
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.

```
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.

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 on it, being careful that you don't move the button within the frame, and select '**Bindings**'. Alternately, you can right click on its entry in the Widget Tree window.

You will see our widget listed at the top of the left panel. We want to bind a left click event to it, so click on the text '**btnGetText**', then click the **Add** button on the menu bar (the far left button). That will open a pop-up window and select **<Button-1>**.

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.

We'll do the same thing with the '**Print Checks**' button, but point to a function called **btnGetChecks_lclick(e)**.

Finally, bind the function '**btnExit_lclick(e)**' to the **btnExit**.

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. The first thing we'll want to examine (but not change) is the routine **set_Tk_var()**. Like all the rest of this file it was generated for us by Page.

```
def set_Tk_var():
    global Colors
    Colors = StringVar()
    global che39
    che39 = StringVar()
    global che40
    che40 = StringVar()
    global che41
    che41 = StringVar()
```

In the above code, there are four global variables defined and set to type **StringVar**. The first, Colors, is the shared variable that binds the RadioButtons together as a group. It is automatically set when one of the RadioButtons is clicked. To see or use the value within the variable use the **.get()** method. You also will usually set one of the RadioButtons at startup of the program as a default value by using the **.set()** method. The other three are for our Checkbox Button widgets.

The other three routines will require a bit of attention.

```
def btnExit_lclick(p1):  
    print('SecondApp_support.btnExit_lclick')  
    sys.stdout.flush()
```

The first one is the routine that is called when the mouse causes the `btnExit_lclick` event to fire. We want the program to exit when this happens (just like in the first example program) so after the line `sys.stdout.flush()`, insert **`destroy_window()`**. It should now look like this:

```
def btnExit_lclick(p1):  
    print('SecondApp_support.btnExit_lclick')  
    sys.stdout.flush()  
    destroy_window()  
  
def btnGetText_lclick(p1):  
    print('SecondApp_support.btnGetText_lclick')  
    sys.stdout.flush()
```

The second routine will print in the terminal window whatever was typed in the entry widget. So, again after the line `sys.stdout.flush()` type `print(w.txtExample1.get())`. This gets the value of the entry widget. We use the `w.` to explicitly refer to the `w` object created in the **`init`** routine that instantiated the user interface.

```
def btnGetText_lclick(p1):  
    print('SecondApp_support.btnGetText_lclick')  
    sys.stdout.flush()  
    print(w.entryExample.get())  
  
def btnPrintChecks_lclick(p1):  
    print('SecondApp_support.btnPrintChecks_lclick')  
    sys.stdout.flush()
```

Last, but not least, we want to print which of the Checkbox Buttons are selected (checked) to the terminal window. This time, we'll have a number of lines of code to enter...

```
if che39.get() == "1": # TWO equal signs on each of the 'if' statements  
    print("chkApples")  
if che40.get() == "1":  
    print("chkBananas")  
if che41.get() == "1":  
    print("chkGrapes")
```

There are two other things that need to be done. First, we need to make sure that when the program starts, none of the check buttons are selected. We'll use a simple routine to accomplish this.

```
def ClearChecks():  
    che41.set("0")
```

```
che40.set("0")
che39.set("0")
```

I put that right after the `set_Tk_var()` routine, but you can put it pretty much anywhere before the line that reads `if __name__ == '__main__':`

Finally, we need to call the set one of the RadioButtons as a default when the program starts. We'll do that and call the `ClearChecks` function from the `init` routine. Right after the line that says `root = top`, insert:

```
Colors.set("Blue")
ClearChecks()
```

By putting it at the end of the `init` routine, these lines of code will be executed AFTER the GUI is created, but before it is shown.

Now, if you are using a version of Page earlier than 4.11, move up to the top of the file and add the following code to the second line:

```
# -*- coding: utf-8 -*-
```

This will help to ensure that Unicode characters are being handled if you enter any into the entry widget. It is a good habit to get into, one which I must admit, forget more often than not. (Thank you Harvard for keeping me on the right track!) In version 4.11, Don has changed Page to automatically do this for us.

Save your program and run it to see how you did. The 'Blue' radio button should be selected and the word "Blue" should be in the sunken label. Also all of the check boxes should be unchecked. Type something in the entry box and click the "Get The Text" button. Whatever you typed in the entry widget should be printed to the terminal window you used to start the program. Check one or more of the check box buttons and click the "Print Checks" button. The text associated with that button(s) should show up in the terminal window.

Good job.

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 2018 is here and Page has matured greatly, I decided to revisit that project and update it a bit.

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 screenshot of the app GUI fresh out of Page.

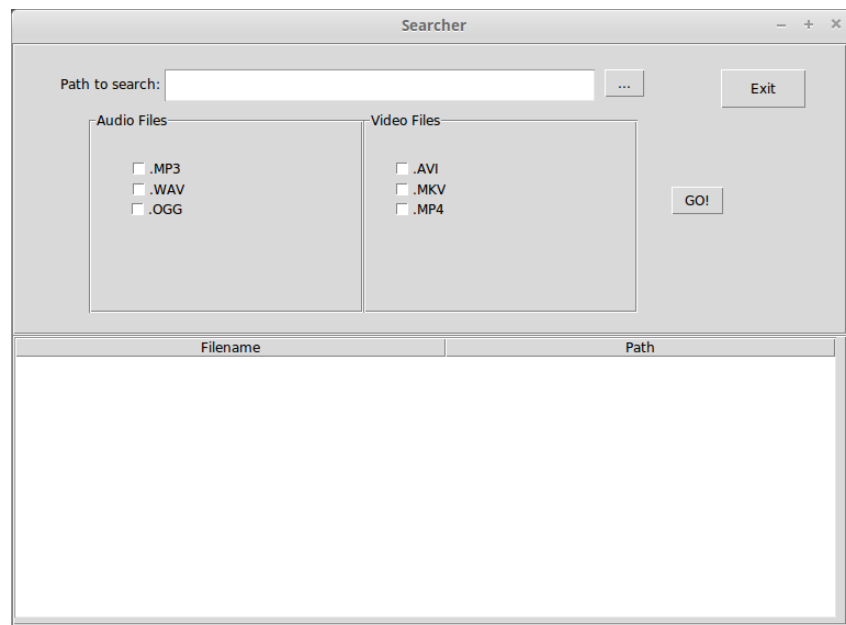


Image 12: Finished Searcher Design UI

Building the UI

We'll start by opening Page and move the new TopMost window 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:

```
alias: txtPath
textvariable: 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.

```
alias: btnSearchPath
command: OnBtnSearchPath
text: “...”
```

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

```
alias: btnExit
text: “Exit”
```

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.

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, 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:

```
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.

```
alias: chkWAV  
anchor: W  
justify: LEFT  
text: '.WAV'
```

variable: **VchkWAV**

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.

alias: **chkAVI**

anchor: **W**

justify: **LEFT**

text: **'.AVI'**

variable: **VchkAVI**

alias: **chkMP4**

anchor: **W**

justify: **LEFT**

text: **'.MP4'**

variable: **VchkMP4**

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:

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.

The Code

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

Remember, if you are using a version of Page prior to 4.11, that you need to put the '# -*- coding: utf-8 -*-' as the second line in the support module as we discussed last chapter.

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.

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

Unfortunately Python3 doesn't handle Tkinter the same way that Python2 does, so we have to import things a bit differently to support both versions of Python.

```
try:
    from Tkinter import *
    import tkMessageBox
    import tkFont
    import tkFileDialog
except ImportError:
    from tkinter import *
    from tkinter import messagebox
    from tkinter import font
    from tkinter import filedialog
```

Another change in Page version 4.11 is in the following section of code. The **py3** variable will be set to True if the version that the code is running under is Python 3.x and False if it is version 2.x. We will

need to check this later on in our code to make sure we are using the correct code between the two different versions of Python.

Page 4.10 and earlier provides this code for us:

```
try:
    import ttk
    py3 = 0
except ImportError:
    import tkinter.ttk as ttk
    py3 = 1
```

And Page 4.11+ provides this code for us:

```
try:
    import ttk
    py3 = False
except ImportError:
    import tkinter.ttk as ttk
    py3 = True
```

The next section of code is the `set_Tk_var()` function. There are no changes here from the Page generated code, but you should see how it's set up for us. This section sets up the global variables that we defined when we created the UI.

```
def set_Tk_var():
    global FilePath
    FilePath = StringVar()
    global VchkMP3
    VchkMP3 = StringVar()
    global VchkWAV
    VchkWAV = StringVar()
    global VchkOGG
    VchkOGG = StringVar()
    global VchkAVI
    VchkAVI = StringVar()
    global VchkMP4
    VchkMP4 = StringVar()
    global VchkMKV
    VchkMKV = StringVar()
```

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 two 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 **destroy_window()** call.

```
def OnBtnExit(p1):
    print('Searcher_support.OnBtnExit')
    sys.stdout.flush()
    destroy_window()
```

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(p1):
    print('Searcher_support.OnBtnGo')
    sys.stdout.flush()
    ClearDataGrid()
    busyStart()
    BuildExts()
    fp = FilePath.get()
    e1 = tuple(exts)
    #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. We have to check the **py3** variable to see if it is set to 1 or 0 (or True or False when using Page 4.11+)

```
def OnBtnSearchPath():
    print('Searcher_support.OnBtnSearchPath')
    sys.stdout.flush()
    if (py3 == 1) or (py3 == True):
        path = filedialog.askdirectory()
    else:
        path = tkFileDialog.askdirectory()
    FilePath.set(path)
```

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

```
def OnTreeviewClick(e):
    print('Searcher_Support.OnTreeviewClick')
    sys.stdout.flush()
    # We will use this in the next chapter.
```

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():
    VchkAVI.set('0')
    VchkMKV.set('0')
    VchkMP3.set('0')
    VchkMP4.set('0')
    VchkOGG.set('0')
    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 allows a **list.clear()** method that could replace this one.

```
def BuildExts():
    del exts[:] # Clear the extentions list, then rebuild it...
    if VchkAVI.get() == '1':
        exts.append(".avi")
    if VchkMKV.get() == '1':
        exts.append(".mkv")
    if VchkMP3.get() == '1':
        exts.append(".mp3")
    if VchkMP4.get() == '1':
        exts.append(".mp4")
    if VchkOGG.get() == '1':
        exts.append(".ogg")
    if 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
    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
    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):
    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 **init()** function is run when the program starts before the main window (Topmost widget) is shown. The first four lines (ending with 'root = top') are generated by Page. The remainder of the function deals with our startup code.

```
def init(top, gui, *args, **kwargs):
    global w, top_level, root
    w = gui
    top_level = top
    root = top
```

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 checkbutton widgets, set the global variable **treeview** to point to our Scrolledtreeview1 widget. Note that we use **w.Scrolledtreeview1**. The '**w.**' refers to our GUI and when we directly make calls to our widgets we have to prepend the '**w.**'. Lastly, we setup the information for our busy cursor functions.

```
# Our code starts here
global treeview, exts, FileList
exts = []
FilePath=StringVar()
FileList=[]
#-----
BlankChecks()
treeview = w.Scrolledtreeview1
SetupTreeview()
#-----
global busyCursor,preBusyCursors,busyWidgets
busyCursor = 'watch'
preBusyCursors = None
busyWidgets = (root, )
```

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'. We also check the **py3** variable to see if we are running under Python3 or Python2 to make the correct call to the font module.

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
        if (py3 == 1) or (py3 == True):
            treeview.column(col, width = font.Font().measure(col.title()))
        else:
            treeview.column(col, width = tkFont.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. Here again, we have to check to see if we are using Python3 or Python2 to make the proper call to the font module.

```
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):
            if (py3 == 1) or (py3 == True):
                col_w = font.Font().measure(val)
            else:
                col_w = tkFont.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?

That's what we are going to do in the next chapter.

Chapter 4 – Linking A Media Player To Searcher

In this chapter, we will be creating a media player for audio and video files, that is linked to the Searcher program we just created.

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

- Scroll bar widgets
- The '.after' method
- Fully event driven programming
- Application Menu Bar
- Adding a second (or more) form to an application.

You'll have to download and install the VLC player library module (python-vlc) as well as the VLC program itself if you don't already have it. You can install the module by **sudo pip install python-vlc**.

The majority of the code and the layout is from the VideoLan.org examples web page https://wiki.videolan.org/python_bindings, and I have made some changes to it to simplify things somewhat.

Here is what the finished project looks like:

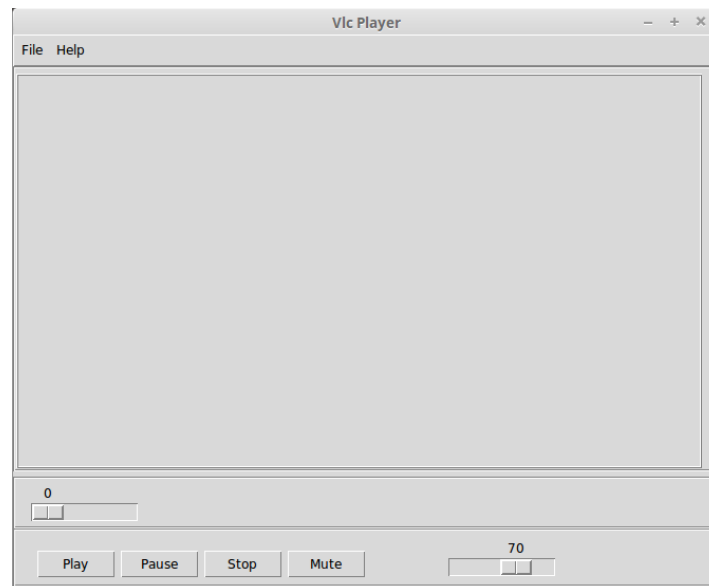


Image 13: VLC Player

Building the UI

Load up Page and you will get your New Toplevel widget. Set the alias to **vlcplayer** and the title to **“VLC Player”** and approximate size and location on the screen. I made mine 670 x 500. Save your project in the same folder that you saved the Searcher project from the last chapter as **vlcplayer**.

Creating the Menu Bar

On the Page main window, select “Widget | Edit Menu Bar...” and you will be presented with a new window that contains the Menu Editor.

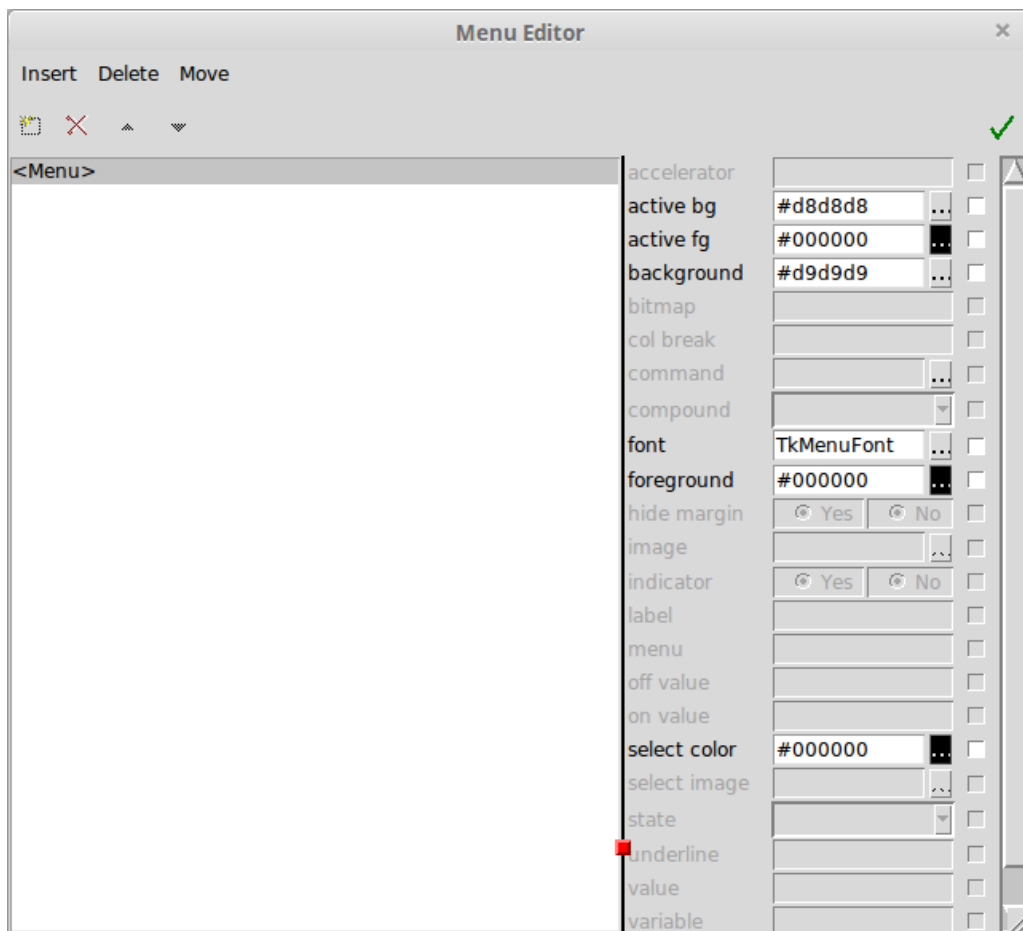


Image 14: Menu Editor

Now, select 'Insert | New Cascade'. This will start the process. Normally, the first menu bar item is for

File operations, like New, Open, Save, Save As, Exit, etc. In this case we will rename the just created item from 'NewCascade' to 'File'. Click on it and in the attributes section of the menu editor, change the label attribute to 'File'. Now select 'Insert | New Command'. Change the label attribute to 'Open', then repeat this process to create a new command titled 'Exit'.

Click again on the 'Open' option and set the command attribute to 'mnuFileOpen'. (You can either change the text directly, or click on the edit button to pop up a window.) Now click on the Exit option and set its command to 'mnuFileExit'.

Next click on '<Menu>' and select 'Insert | New Cascade'. This will create a new "top level" menu item underneath the File menu set. Rename this from 'NewCascade' to 'Help'. Now select 'Insert | New Command' and rename it to 'About'. Set the command attribute to 'mnuHelpAbout'.

Click on the green check mark on the top right to close the Menu Editor. Save your tcl project file, generate the python code module and the support module.

If you look at the support module, you will see that Page has generated the three menu functions (mnuFileOpen(), mnuFileExit() and mnuHelpAbout()) for us.

Next, we need three frame widgets. The first, will hold a canvas widget that will act as the video “screen” when we play videos. The second will hold a slider widget for positioning within the media file. It will automatically update as the media plays. The third, will hold four buttons and a slider widget.

The first frame widget should be about $\frac{3}{4}$ of the Toplevel widget vertically and fill the space horizontally. Name it 'frameVideo'. The next one should take half of the remaining vertical space and fill the space horizontally as well. Name it frameSlider. The final one should be about the same size as the second and name it frameButtons.

Now place a canvas widget into the frameVideo widget. You can use the default alias name (Canvas1). You want to make it fill the entire frame.

In the frameSlider, place a horizontal slider widget. Set the attributes for it as follows:

alias: **TimeSlider**
command: **ScaleSel**
length: **500**
to: **1000**
variable: **ScaleVar**

Next in the frameButtons widget, you need to place four button widgets and a horizontal slider along the same horizontal line. Each button should be 75 pixels wide and spaced equally. Here are the

attributes for the four buttons from left to right:

alias: **btnPlay**
text: **Play**

alias: **btnPause**
text: **Pause**

alias: **btnStop**
text: **Stop**

alias: **btnMute**
text: **Mute**

And the attributes for the horizontal slider:

alias: **VolumeSlider**
command: **VolumeSel**
variable: **VolumeVar**

If you want, you can put a label between the **btnMute** and the **VolumeSlider** with the text of “Volume:”, but this is strictly optional.

Finally, we need to bind the mouse button-1 to each of the four buttons to:

btnPlay: **OnBtnPlay**
btnPause: **OnBtnPause**
btnStop: **OnBtnStop**
btnMute: **OnBtnMute**

Now we can put in our code.

The Code

The operation of this program is two fold. First, it can be launched as a standalone program and by use of the menu bar, you can select a media file to be played. In the second way, it will be called, through some minor modifications, from our Searcher program which will pass the file path and filename to the player program and start the playing the media file.

We'll be working with the support file. The first thing we want to do is setup the imports, so change what page gave us to:

```
# Import the VLC library
import vlc
# Import the standard libraries
import sys
import os
import pathlib
import time
import platform

try:
    from Tkinter import *
    import tkFileDialog
    import tkMessageBox
except ImportError:
    from tkinter import *
    from tkinter import messagebox
    from tkinter import font
    from tkinter import filedialog

try:
    import ttk
    py3 = 0
except ImportError:
    import tkinter.ttk as ttk
    py3 = 1
```

We need to add to the init function that has already been created for us. We will add the lines that are in bold below the line **root = top**:

```
def init(top, gui, *args, **kwargs):
    global w, top_level, root
    w = gui
    top_level = top
    root = top
    global Instance, player
    global timeslider_last_val, timeslider_last_update
    timeslider_last_val = 0
    timeslider_last_update = 0
    Instance = vlc.Instance()
    player = Instance.media_player_new()
    if len(args) != 0:
        temp = list(args)[0]
        dirname = temp[0]
        filename = temp[1]
        Media = Instance.media_new(str(os.path.join(dirname, filename)))
        player.set_media(Media)
        if platform.system() == 'Windows':
            player.set_hwnd(self.GetHandle())
        else:
            player.set_xwindow(GetHandle())
        OnBtnPlay()
    else:
```

```

    OnOpen()
    VolumeVar.set(70)

```

We are checking for the length of the args variable that is passed into the init function. We are doing this to allow the program to be run either in a standalone method or called by the searcher project we did in the last chapter. When the media player is called from searcher, it will pass the directory name and filename of the media file to be played through the *args variable. We also need to do much of the work that is done in the **OnOpen()** function, so we set the handle to the canvas widget before we call the **OnBtnPlay()** function.

Next we will start working on our button functions. Notice we use the **.config()** method to modify the text attribute of the mute button.

```

def OnBtnMute(p1):
    sys.stdout.flush()
    is_mute = player.audio_get_mute()
    print("is_mute = {0}".format(is_mute))
    if is_mute == 1:
        w.btnMute.config(text = 'Mute')
    else:
        w.btnMute.config(text = 'Unmute')
    player.audio_set_mute(not is_mute)
    VolumeVar.set(player.audio_get_volume())

```

Here we simply call the **player.pause()** function to pause and unpause the player.

```

def OnBtnPause(p1):
    sys.stdout.flush()
    player.pause()

```

The **OnBtnPlay()** function will call the **OnOpen()** function to get things going. We also use the **.after** rather than a simple loop or a separate threaded timer. Tkinter uses the **root.after** method to create a timer for us, however it isn't quite as accurate as a separate timer, but is accurate enough to handle the things we need. We get the **Timer_id** value when we call the initial **root.after(time,function)** method to get things set up. Since it's the first time we call it, we use 0 as our time and set the function to call to our **OnTick()** function. Once we get into the **OnTick()** function, the last line sets the time for the next call, which in our case is about 1000 milliseconds.

```

def OnBtnPlay(p1=None):
    sys.stdout.flush()
    global Timer_id

    if not player.get_media():
        OnOpen()
    else:
        # Try to launch the media, if this fails display an error message
        if player.play() == -1:

```

Learning Page - A Python GUI Designer

```
        errorDialog("Unable to play.")
    Timer_id = root.after(0, OnTick())
```

When we click the stop button, we call the **player.stop()** function and set the time slider position to 0.

```
def OnBtnStop(p1):
    sys.stdout.flush()
    player.stop()
    # reset the time slider
    w.TimeSlider.set(0)
```

The File | Exit menu command calls **destroy_window()** as we have done many times before.

```
def OnMnuFileExit():
    sys.stdout.flush()
    destroy_window()
```

From the File | Open menu command, we call the **OnOpen()** function to get things going when in the standalone mode.

```
def OnMnuFileOpen():
    sys.stdout.flush()
    OnOpen()
```

We will flesh out the Help | About menu command later on, but I put it here simply for completeness.

```
def OnMnuHelpAbout():
    sys.stdout.flush()
```

The **ScaleSel()** function allows the user to set the gross position in the media file to act as a fast-forward/rewind analogue. The user must click and drag the slider “button” rather than clicking within the tray to make it “jump” to the mouse position, however if the user holds mouse button-1 down within the tray, the slider will jump in steps quickly. We assign the **nval** variable to the **ScaleVar** variable using the **.get()** method, then compares the current value to the last value (**timeslider_last_val**) to minimize the possibility of an endless loop. If they are different, we update the last time the slider was updated, convert the position to milliseconds and update the player position with the **set_time()** function.

```
def ScaleSel(*args):
    global timeslider_last_update, timeslider_last_val
    sys.stdout.flush()
    if player == None:
        return
    nval = ScaleVar.get()
    sval = str(nval)
    if timeslider_last_val != sval:
```

```

timeslider_last_update = time.time()
mval = "%.0f" % (nval * 1000)
player.set_time(int(mval)) # expects milliseconds

```

The **VolumeSel()** function (as the name suggests) allows the user to change the volume of the file being played. This function is called whenever the volume slider is moved. If the player hasn't been started, the function simply is exited. Otherwise, we get the integer value of the slider (via the **VolumeVar** variable) and assign it to a variable called **volume** for simplicity. We then “normalize” the value and call the player's **audio_set_volume()** function.

```

def VolumeSel(*args):
    if player == None:
        return
    volume = int(VolumeVar.get())
    if volume > 100:
        volume = 100
    elif volume == 0:
        volume = 70
    resp = player.audio_set_volume(volume)

```

The **OnOpen()** function sets the filename and path of the media file to be played by using an **askopenfilename** style file dialog provided by Tkinter.

```

def OnOpen():
    p = pathlib.Path(os.path.expanduser("~"))
    if (py3 == 1) or (py3 == True):
        fullname = filedialog.askopenfilename(initialdir = p, title = "choose your
file", filetypes = (("all files", "*..*"), ("mp4 files", "*.mp4")))
    else:
        fullname = tkFileDialog.askopenfilename(initialdir = p, title = "choose your
file", filetypes = (("all files", "*..*"), ("mp4 files", "*.mp4")))
    if os.path.isfile(fullname):
        dirname = os.path.dirname(fullname)
        filename = os.path.basename(fullname)
        print("{0} - {1}".format(dirname, filename))
        Media = Instance.media_new(str(os.path.join(dirname, filename)))
        player.set_media(Media)
        if platform.system() == 'Windows':
            player.set_hwnd(self.GetHandle())
        else:
            player.set_xwindow(GetHandle())
        OnBtnPlay()

```

Here is our “timer” callback function **OnTick()**. We use this to do some maintenance to the various variables and widgets. Since the “published” length of a media file can change during playing, we update the length of the slider by setting the “to” attribute to the length of the file gathered by a call to the player's **get_length()** method and update the current position on the time slider.

Learning Page - A Python GUI Designer

```
def OnTick():
    global timeslider_last_update, timeslider_last_val, Timer_id
    if player == None:
        return
    # since the self.player.get_length can change while playing,
    # re-set the timeslider to the correct range.
    length = player.get_length()
    dbl = length * 0.001
    w.TimeSlider.config(to=dbl)
    # update the time on the slider
    tyme = player.get_time()
    if tyme == -1:
        tyme = 0
    dbl = tyme * 0.001
    timeslider_last_val = ("%0f" % dbl) + ".0"
    # don't want to programatically change slider while user is messing with it.
    # wait 2 seconds after user lets go of slider
    if time.time() > (timeslider_last_update + 2.0):
        w.TimeSlider.set(dbl)
    Timer_id = root.after(1000, OnTick)
```

The **GetHandle()** function allows us to get the widget ID of the canvas widget, which is used by the **OnOpen()** function so that the player knows where to render the video of the media file.

```
def GetHandle():
    return w.Canvas1.wininfo_id()
```

Finally, we have the **errorDialog()** function that takes whatever message we want to be displayed to the user as a **showerror** style **messagebox**.

```
def errorDialog(errormessage):
    tkMessageBox.showerror('Error', errormessage)
```

The About Box

While Tkinter has a number of dialog boxes created for us already, strangely they don't have an about box. So, I've created a generic one that I will present here. Below is a screen shot of the finished About Box used in the Video Player program. You can see, there are four label widgets that are set with a sunken relief attribute one above the other, one ScrolledText widget and a button widget to dismiss the About Box when we want it to go away.

The About Box Form

When creating the About Box, be sure to save the code into the same folder you are using for the media player project, which should be the same as the searcher project.

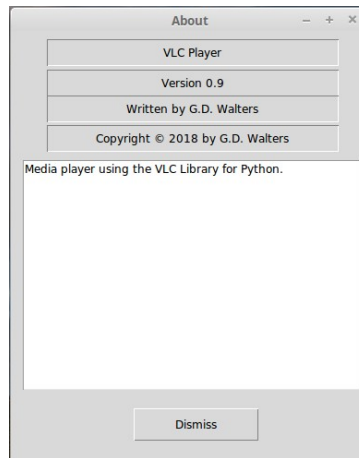


Image 15: The Generic About Box

Make the TopMost widget somewhat rectangular and placed in the middle of the screen. Set the Title attribute to **“About”** and the alias to **“About”** as well.

Now place four label widgets into the TopMost widget, near the top one above the other, equally spaced vertically and all having the same width. Set the **relief** attribute to **“sunken”** for all four. I didn't change the text for any of them, since each will have their text attributes changed at run time. Name them, from top to bottom as:

lblProgName, lblVersion, lblAuthor and lblCopyright.

Next, place a ScrolledText widget taking up the lion's share of the TopMost widget (leaving room for a fairly large button near the bottom). The only attribute that really needs to be changed here is the “wrap” attribute, which need to be set to **“WORD”**. However, you have to click on the “Text” under the Scrolledtext entry in the Widget tree to be able to get to it. While good programming practice suggests that we set the alias of the widget to something besides the default, I didn't bother.

Finally, we need a standard button widget centered horizontally and vertically in the remaining space and somewhat large. This will be our Dismiss button. Set its alias to **btnDismiss**, set the text attribute to **“Dismiss”** and bind a mouse button-1 event to point to the function **OnBtnDismiss**. Save the project and both code modules with the name **“GenAboutBox”**.

The About Box Code

The code for the About Box is really very simple with a couple of important items.

Learning Page - A Python GUI Designer

It is very important that you add the following line to the top of your support file if it is not there already.

```
# -*- coding: utf-8 -*-
```

If you don't you will get the dreaded “**SyntaxError: Non-ASCII character '\xe2' in file...**” error when you try to start the generic about box, since we are using a Unicode character to support the “©” symbol.

Here's the standard imports.

```
import sys

try:
    from Tkinter import *
except ImportError:
    from tkinter import *

try:
    import ttk
    py3 = 0
except ImportError:
    import tkinter.ttk as ttk
    py3 = 1
```

In the **OnBtnDismiss** function, we use **root.withdraw()** to *hide* the about box rather than the **destroy_window()** that we normally use to close the program. If we were to use the **destroy_window()** here, it would close the entire program.

```
def OnBtnDismiss(p1):
    print('GenAboutBox_support.OnBtnDismiss')
    sys.stdout.flush()
    root.withdraw()
```

In the **LoadForm()** function, we will (as the function name suggests) load all the information sent from our video player into the labels and the ScrolledText widget. For the labels, we use the **.configure(attribute =)** format to set the label text. For the ScrolledText widget, we use the **.insert()** method to place the text into the text box at the end of whatever is already there, which in this case is nothing.

```
def LoadForm():
    global temp
    copyright_symbol = u"\u00A9"
    w.lblProgName.configure(text = temp[0])
    w.lblAuthor.configure(text = "Written by {0}".format(temp[1]))
    w.lblVersion.configure(text = "Version {0}".format(temp[2]))
    w.lblCopyright.configure(text = temp[3])
    w.Scrolledtext1.insert(END, temp[4])
```

The init function is pretty much the same as always, except we are adding two lines. One for getting the

data sent from the media player program which includes 5 items (Program Name, Author name, Program Version, Copyright notice and the program Info) as well as the call to LoadForm.

```
def init(top, gui, *args, **kwargs):
    global w, top_level, root, temp
    w = gui
    top_level = top
    root = top
    temp = list(args)[0]
    LoadForm()
```

The **destroy_window** and the **if __name__** functions are the standard that are already created for us by Page without any changes, but I've included them just to be complete.

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

if __name__ == '__main__':
    import GenAboutBox
    GenAboutBox.vp_start_gui()
```

Linking The About Box To The Media Player

Now that all the hard work has been done, we need to link the About Box program files to our Media Player. This is very simple and only requires a few of lines of code to be added to the Media Player support file.

First, we need to add a line somewhere that imports our About Box project as a library. I like to put it into the top of the support file with the rest of the imports, just to keep them all together. So, under the line that says **import platform**, insert:

```
import GenAboutBox
```

Now in the function, we need to add the lines that define the variables that we will pass on to the About Box program. Here's the full function including the lines that Page created for us.

```
def OnMnuHelpAbout():
    print('vlcplayer_support.OnMnuHelpAbout')
    sys.stdout.flush()
    # Our code from here down
    copyright_symbol = u"\u00A9"
    ProgName = "VLC Player"
    Author = "G.D. Walters"
    Version = '0.9'
```

Learning Page - A Python GUI Designer

```
Copyright = "Copyright " + copyright_symbol + " 2018 by G.D. Walters"
Info = "Media player using the VLC Library for Python."
GenAboutBox.create_About(root, [ProgName, Author, Version, Copyright, Info])
```

Breaking it down, we define the “©” symbol to the `copyright_symbol` variable as a Unicode character. Next we define the rest of the variables (`ProgName`, `Author`, `Version`, `Copyright` and `Info`) by assigning strings to them to get them ready to pass to the `create_About` function of the About Box program. Finally we call the **`create_About`** function (the entry point within our next program to show) with the appropriate variables, our data being passes as a list. Now, let's take a look at the `create_About` function in the `GenAboutBox.py` file, which was created for us by Page.

```
def create_About(root, *args, **kwargs):
    '''Starting point when module is imported by another program.'''
    global w, w_win, rt
    rt = root
    w = Toplevel (root)
    top = About (w)
    GenAboutBox_support.init(w, top, *args, **kwargs)
    return (w, top)
```

This function exists in any main python file that Page creates and is used to allow us to have multiple form programs. When this function is called, it in turn calls the **`.init()`** function that is in the support file, which is loaded before the program shows the main window of the additional form, in this case the About Box.

Notice the call passes **`*args`** to the **`.init()`** function. This is the list of variables that will be passed to fill the label widgets and the Scrolledtext widget. When the **`.init()`** gets these variables, they can be cast into a list that can then be accessed as we do in the `LoadForm` function within the support module of the About Box.

This allows your program to have multiple forms and pass information between them.

Linking The Media Player To The Searcher Program

Now we have to modify the Searcher program, pretty much as we did when we added the About Box to the media player.

In the support module for Searcher, we need to import the media player program, so add

```
import vlcplayer
```

at the top with the other imports. Next add the following line to the **`OnTreeviewClick(e)`** function.

```
row = treeview.identify_row(e.y)
col = treeview.identify_column(e.x)
```

```
filename = treeview.set(row,0)
path = treeview.set(row,1)
vlcplayer.create_VLC_Player(root,[path,filename])
```

Here, we get the row and column that the mouse button-1 click occurred in then we use that information to set the path and filename variables from the information in the grid of the ScrolledTreeView widget. Finally we call the create_VLC_Player function with the two variables.

That's about it. Save all your files and make sure that clicking on all the menu items and buttons that everything work as expected.