Python GUI Programming with PAGE

Create professional-looking GUIs for Python applications efficiently and effectively

Gregory Walters

Python GUI Programming with PAGE

Create professional-looking GUIs for Python applications efficiently and effectively

Gregory Walters

www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-378

www.bpbonline.com

www.spbonline.com

Dedicated to

My son Douglas

About the Author

Gregory Walters has been programming since 1972. Since 2009, he has been writing articles about programming in Python for Full Circle Magazine. To date, he has done well over 100 Python articles for them as well as book reviews and other things. In addition, he has been using PAGE since about the same time. He has also written a 4 chapter tutorial on using PAGE that has been included in each copy of the Page distribution since early 2019.

Greg has written one book on Python, co-authored one book on GANs in Python and done Technical Review on over 20 books for various publishers.

Greg is currently semi-retired and live in Central Texas, U.S.A. In his spare time, he runs the "Unofficial Page Support Page" on Discord, enjoys cooking, playing music and building his own musical instruments.

Acknowledgements

There are a few people I want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, I would like to thank my son for continuously encouraging me for writing the book, put up with me when things weren't going the way I expected — I could have never completed this book without his support.

I am grateful to Don Rozenberg the author of PAGE. Without his support, guidance and willingness to accept suggestions, I would never have been able to create this book.

I also want to thank Ronnie Tucker, the Editor in chief of Full Circle Magazine, who back in 2009, gave me my first major publication attribution. Without his constant backing and support over the years, I wouldn't be where I am today.

To my dear friend Halvard in Norway, who has for many years sent me kind words, gentle persuasion, and positive reinforcement on a constant basis in all that I've done.

My gratitude also goes to the team at BPB Publication for keeping me on track during the difficult process of pulling this book together.

Preface

Learning PAGE is designed to take the reader from installation of the PAGE designer program all the way to creating complex Graphical User Interfaces for Python programs. PAGE has been around many years but unfortunately has not gotten much "press" in the media, so it is not very well known.

This book is designed to take the reader from installation of the Page designer through to being able to create complex GUI interfaces. These include front end interfaces for SQLite databases (also extends to other databases) to extending the GUI to use the Canvas widget to include drawing rectangles, ellipses, text and other things. Topics include using standard Tk widgets (buttons, static text, entry, etc.) to the ttk toolkit and expanding the capabilities of each. Also covered is using third party widget libraries and creating a custom widget and including it in real-world programs.

This book is divided into 12 chapters plus the Introduction. They will cover the basics of using PAGE, adding widgets to the form and changing the attributes that those widgets have and the process of generation of the Python modules. After that, the process of modifying the Python files to create the end program.

Introduction – will show the user how to install and more importantly how to properly start PAGE.

Chapter 1: Introduction to PAGE – will show the various windows that make up PAGE, how to create the user's first program using PAGE. The widgets used for this project will include Radiobuttons, Frames, dynamic and static Labels and more.

Chapter 2: Going Further – will expand the learning process to include two projects, one showing how to create a program that has multiple forms and how to create splash screens.

Chapter 3: Standard Tk Widgets – will cover the PAGE standard Tk library of

widgets, and the attributes that each possess.

Chapter 4: The Pinger Program – will show how to create a real world program that uses the ping internet utility. This project shows how to control the Label colors dynamically and basics of using the Tk Text widget.

Chapter 5: Using Graphics – covers the use of graphical images in the reader's projects, including which widgets support the use of graphics and how to embed graphics into the project both by way of the designer and in code as well.

Chapter 6: Menus and Popup Menus – goes through the use of Menu bars, Button Bar menus and popup (or context) menus.

Chapter 7: Using ttk Widgets – similar to Chapter 3 which explored the all the standard Tk widgets, this chapter explores all the ttk group of widget that expand the widget set and provide the ability to create more modern looking programs.

Chapter 8: Custom Controls – explores the ability to use third party widgets, not normally supported by PAGE, the enhance the look, feel and usability of the reader's programs.

Chapter 9: Creating a SQLite Database Front End – covers the ability to use a PAGE project to act as a front end for advanced Database programs to display, enter and save information back to the database.

Chapter 10: Creating Custom Profiles – covers the Preferences editor to allow the user to control the look and feel of PAGE itself and the projects it creates.

Chapter 11: Using the Canvas Widget – shows how the Tk Canvas widget, which is one of the most complex and powerful widgets in the Tk toolkit, can be controlled to create graphs, text, lines, ovals and more.

Chapter 12: Conclusion – wraps up the book touching on getting support and further resources for the reader to expand their knowledge of Tkinter programs and PAGE.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the *Coloured Images* of the book:

https://rebrand.ly/a766f8

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Python-GUI-Programming-with-PAGE**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version

at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Introduction 1
Why did I write this book? 2
What this book is 2
What this book is NOT 3
What you need to know before reading this book 3
Text conventions used 3
Code blocks 3
Terminal commands 4
Inline references 4
Widget attributes 4
Notes, tips and warnings 4
Requirements 5
Windows 5
Linux 5
Raspberry Pi (Linux) 5
OSX 5
Installation 6
Windows 6
Linux 6
Raspberry Pi (Linux) 6
OSX 7
Starting PAGE 7
Getting support 9
Last thoughts 9

1. Introduction to PAGE 11
Introduction 11
What you will learn 11
Starting page 12
The PAGE designer forms 13
The Main Window 13
The Attribute Editor 13
The Widget Tree 15

The Widget toolbar 15
The Designer form 16
The Layout of the windows 17
GUI design 17
The Code 27
Conclusion 31
Multiple Choice Questions 32
Answers 32
Questions 32
Answers to the questions 33
Key Terms 33

2. Going Further 35
Introduction 35
What you will learn 35
Section 1 – Multiple forms 36
Designing the GUI 36
The code 44
Section 2 – Creating a Splash Form 50
What you will learn: 50
Designing the GUI 51
The Code 54
Conclusion 58
Multiple Choice Questions 58
Answers 58
Questions 59
Answers to the questions 59
Key terms 59

3. Standard Tk Widgets 61
Introduction 61
What you will learn 61
Tk Widgets 62
Common attributes 63
Geometry 65
Toplevel 65

Button 66
Canvas 68
Checkbutton 68
Entry 70
Frame 71
Label 72
Labelframe 73
Listbox 74
Message 74
Radiobutton 75
Scales 75
Spinbox 76
Text 77
Popupmenu 78
All Tk widgets in action 78
Conclusion 80
Multiple Choice Questions 80
Answers 80
Questions 80
Answers to the questions 81
Key terms 81

4. The Pinger Program 83
Introduction 83
What you will learn 83
The History of Pinger 84
Designing the GUI 85
Adding the main menu 85
Filling out the rest of the Main GUI 89
Adding the About form 92
The code 94
Conclusion 130
Multiple Choice Questions 130
Answers 131
Questions 131

Answer to the questions *131*
Key terms 132

5. Using Graphics 133
Introduction 133
What you will learn 134
Before you start 134
Getting quality graphics 135
Converting image formats *135*
Creating the GUI 136
Writing the Code 142
Conclusion 146
Multiple Choice Questions 146
Answers *146*
Questions 147
Answers to the questions *147*
Keywords 147

6. Menus and Popup Menus 149
Introduction 149
What you will learn 149
Menu overview 150
Before we get started 151
Possible menu items 151
Radiobuttons *151*
Checkbuttons *152*
Bindings *154*
Accelerators *154*
Creating the GUI 155
Writing the code 166
Conclusion 177
Multiple Choice Questions 177
Answers *178*
Questions 178
Answers to the questions *178*
Keywords 179

7. Using ttk Widgets 181
Introduction 181
What you will learn 182
What is a ttk widget 182
What is a theme? 182
What is a style? 183
Creating a simple custom style 186
Getting more themes 189
Special ttk widgets 190
TCombobox widget 190
TNotebook 191
TProgressbar 191
TSeparator 192
TSizegrip 192
Treeview 192
Conclusion 193
Multiple Choice Questions 193
Answers 193
Questions 194
Answers to the questions 194
Key terms 194

8. Custom Controls 195
Introduction 195
What you will learn 195
Custom Control Widget 196
Project #1 – tkDatePicker 196
Designing the program 197
Project #1 Code 197
Project #2 – ScrolledCheckedListbox 202
Designing the program 203
Project #2 Code 204
Project #3 – Plotext 209
Designing the program 212
Project #3 Code 213

Conclusion 221
Multiple Choice Questions 222
Answers 222
Questions 222
Answers to the questions 223
Key terms 223

9. Creating a SQLite Database Front End 225
Introduction 225
What you will learn 226
Things you will need 226
Getting started – SQLite databases at a glance 227
Creating the project database 228
The GUI form 232
Writing the Code 234
Conclusion 258
Multiple Choice Questions 258
Answers 259
Questions 259
Answers to the questions 259
Key terms 259

10. Creating Custom Profiles 261
Introduction 261
What you will learn 262
Getting started 262
The Basics tab 264
Font tab 266
Colors tab 268
Saving the preferences 271
Creating a custom preference profile 271
Conclusion 272
Multiple Choice Questions 272
Answers 273
Questions 273

Answers to the questions 273

11. Using the Canvas Widget **275**
Introduction 275
What you will learn 277
Project #1 – Canvas functions 277
Code for Project #1 ..... *278*
Project #2 – animated canvas 284
How it works *285*
Conclusion 295
Multiple Choice Questions 295
Answers *295*
Questions 295
Answers to the questions *296*

12. Conclusion **297**
Going further 297
Resources 298

Index **299-303**

Introduction

PAGE is a free and open source that creates Graphical User Interfaces for Python programs. It allows a user to create these GUIs in a **Rapid Application Development (RAD)** mindset. PAGE uses Tcl/Tk to create the GUI and then saves the code in native Python/Tkinter files. It has been created and maintained by Don Rozenberg for many years.

PAGE enjoys over 2,000 downloads per month and is used worldwide. You do not need to understand Tcl/Tk to use PAGE. You do need to have a basic knowledge of Python, however.

I have been writing about and using PAGE since about 2011. Back then, PAGE was about the only thing that allowed Python users a quick and easy way to utilize the Tkinter interface easily, without having to be very knowledgeable about the Tk toolkit. There were many books and websites about WxPython/WxWindows and the Qt toolkit, but those had some rather complicated installation instructions and were not completely free and open source. Some confusing licensing restrictions caused many Python users to shy away from any type of GUI design. Tkinter comes with Python and is the de facto standard GUI package.

The current version of PAGE, as of this writing, is version 7.6.

Why did I write this book?

For the majority of my programming career, I have been a mentor for other programmers in the various languages that I have used. I have been considered the kind of programmer that thinks "outside the box", finding unique solutions for difficult problems.

When it comes to PAGE, I have been using it for almost 10 years now and have enjoyed a close relationship with *Don Rozenberg*. I have been blessed to be one of the people whom Don relies on to test (and attempt to break) new versions of PAGE. Because of this, I have been able to learn a great deal about how PAGE works and how to avoid certain limitations that are not quite obvious to others.

Most of the projects that I will be presenting in this book are either projects that I have created for other users to address their questions about "How do I ..." or projects that I have created for myself, either for my Full Circle articles or my own programs.

What this book is

This book is designed to teach a PAGE user how to use the program to create some complex GUI frontend programs in Python.

It is designed to take the user from beginner to, by the end of the book, knowing how to use most of the Tkinter widgets at a normal programming level.

I have broken the book into 4 sections:

1. **Beginning:** This section is designed to get you started using PAGE from installation to simple programs using some of the widgets and concepts that you use most often.
2. **PAGE in depth:** This section will discuss the standard widgets from Tk and ttk and some real-world projects that include dynamically controlling widget attributes, menus and general use of graphics.
3. **Advanced Concepts:** This section will show how to use third-party widgets and create your own custom widgets to extend the abilities of PAGE.
4. **Summary:** Going through what has been learned, some tips and tricks, and

4. Summary: Going through what has been learned, some tips and tricks, and some thoughts about best practices.

What this book is NOT

This book is not designed to teach any given coding style. Coding styles are, in my opinion, a deeply personal subject and more times than not, will be dictated by the organization that you work for at the moment. My programming style is slanted to that of one that teaches the basic "how to", not necessarily the "best" or "only" way to do things. It is not going to show how to write everything in a single line of code. It uses a large number of comments, more for the teaching aspect than anything else.

This book is not designed to cover everything that can be done with PAGE or Tkinter. It is designed to cover, at a gross level, all the normal widgets that PAGE supports (which is most of them).

What you need to know before reading this book

You should have a basic knowledge of Python. You DO NOT need to be a Python expert or a data scientist. You should have a basic knowledge of variables and their use and creating functions. Anything else will be helpful, but I have tried very hard to create and document reasonable projects that will provide you with everything you need to be able to, by the end of the book, create your own advanced GUI front ends for almost any program you want to create in Python.

Text conventions used

There will be many different types of text that you should know about.

Code blocks

When I am presenting blocks of code, I will use a Consolas font, like this:

```
try:
    import Tkinter as tk
except ImportError:
```

```
import tkinter as tk
```


If I need to draw attention to a specific line of code within a block, I will make that line bold:

```
def on_btnExit():
    print('secondaryform_support.on_btnExit')
    sys.stdout.flush()
    destroy_window()
```

Terminal commands

Whenever I need you to type a command in a Linux terminal or Windows command prompt, it will be in a bold consolas font. It will always be preceded by a "$" for Linux Terminals or ">" for Windows command prompts as follows:

```
$ tar zxf /path/to/ActiveTcl-download.tar.gz
```

or

```
> c:\page\winpage.bat
```

Inline references

Sometimes, I will give filenames or information within a sentence. When this happens, I will again, use a bold monospace font within the sentence. Like this:

"Now, save the file as **`myfirstpage.tcl`** to your working directory."

Widget attributes

All the projects we will be creating will use widgets or controls and there will be various settings or attributes that will need to be modified. To make it easy for you, I will use a table format to show the attribute name in the left column and the value of the attribute in the right. Something like this...

alias	btnExit

command	on_btnExit
text	Exit

Notes, tips and warnings

Whenever I need to provide you with some special information, I will place the text within a box.

Note: When entering the callback function for the `root.after` *command, be sure NOT to put* `parentheses` *at the end of the function name as you normally would. Be sure to format it as follows:*

```
Timer_id = root.after(0, on_tick)
```

Requirements

Windows

Under Windows, you will need, of course, Python 3.8 or greater. I will be using Python 3.8.10. Earlier versions of PAGE required you to have Tcl/Tk installed on your system, but as of PAGE 6.0, this is no longer required. So, the only actual requirement for PAGE is Python 3.8 or greater.

You will also need some sort of an editor to keep up with your Python code. There is a very nice free cross-platform **Integrated Development Environment (IDE)** called Geany. I will be using VS Code, which is also free.

Linux

While Linux usually has everything you need, as with Windows, you need Python 3.8 or greater.

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

Raspberry Pi (Linux)

You will need to use the latest Raspberry Pi OS. This should include Python 3.8 or greater, so you should be ready to go.

OSX

Since I do not have access to any OSX machines, I really cannot speak about 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 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 just as simple as downloading the PAGE package, extracting it and creating an alias for PAGE.

Download the PAGE distribution at **https://sourceforge.net/projects/page/**. The page normally detects your OS, so it should prompt you to download the tar-gzipped latest version.

Unpack the downloaded file into your home directory. Usually, the File manager will allow you to do this by right clicking on the file and selecting "`Open with Archive Manager`".

Remove any "`.pagerc`" files in your root folder.

If you are running a Linux distribution based on Debian, like Ubuntu or Linux Mint, edit your `~/.bashrc` file to include an alias to your PAGE file. Look for a line that says "`# some more ls aliases`". I keep my PAGE folder under the Downloads folder, so I would enter:

```
alias page='python /home/greg/Downloads/Page-7.6/page/page.py'
```

Be sure to save the file. Finally, either close that terminal or within the terminal type:

```
$ source ~/.bashrc
```

This forces the terminal to reload the resource file. Now, whenever you want to start PAGE, all you have to do is change to your project folder and type "page".

Raspberry Pi (Linux)

Make sure that you have the latest Raspberry Pi OS build. Once you have your Raspberry Pi running, you can follow the given steps:

1. Download the PAGE distribution from **https://sourceforge.net/projects/page/**.

2. Once it is downloaded, untar the downloaded file into a convenient directory.
3. Open a terminal window and navigate to that folder.
4. Finally, edit your ~/.bashrc file to add an alias to create an easy command that will start PAGE from your terminal:

```
$ mousepad ~/.bashrc
```

look for a line that says **"# some more ls aliases"**

5. Below this line, add a blank line and type:

```
alias page='python /home/greg/Downloads/Page-7.6/page/page.py'
```

6. Save the file. Now, you have a choice. You can close and reopen your terminal or you can force the terminal to reload the **bashrc** file.

```
source ~/.bashrc
```

You are done with the installation.

OSX

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

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

As under Windows and Linux, create a "master directory" to hold your development files.

Starting PAGE

One of the frequently asked questions is about having problems embedding graphics in various widgets. It works fine in the designer, but when the user tries to run the generated program, they receive an error message about missing graphics. Almost

every time, this is due to not starting PAGE correctly, in whichever operating system you are using. It is **VERY** important to start PAGE directly from the folder that you plan to save your project into.

This is why I always suggest you create a development folder, whether for learning or actual work. For my own work, I have numerous folders for this. One I call **POC** for **P**roof **O**f **C**oncept programs. It is a kind of a sandbox for ideas that I have no idea if it will actually work or not. If things work out correctly and I like the results, I will move the working folder into my **Development** folder. This is where I keep any projects that I expect to get to at least a Beta state. At that point, I either move or copy to my local Github project folder so I can have source code control.

Using a Windows system for a quick example, I put my development folder directly in the `C:\` folder. (This is where PAGE installs all its files.) So, my directory tree would look something like this:

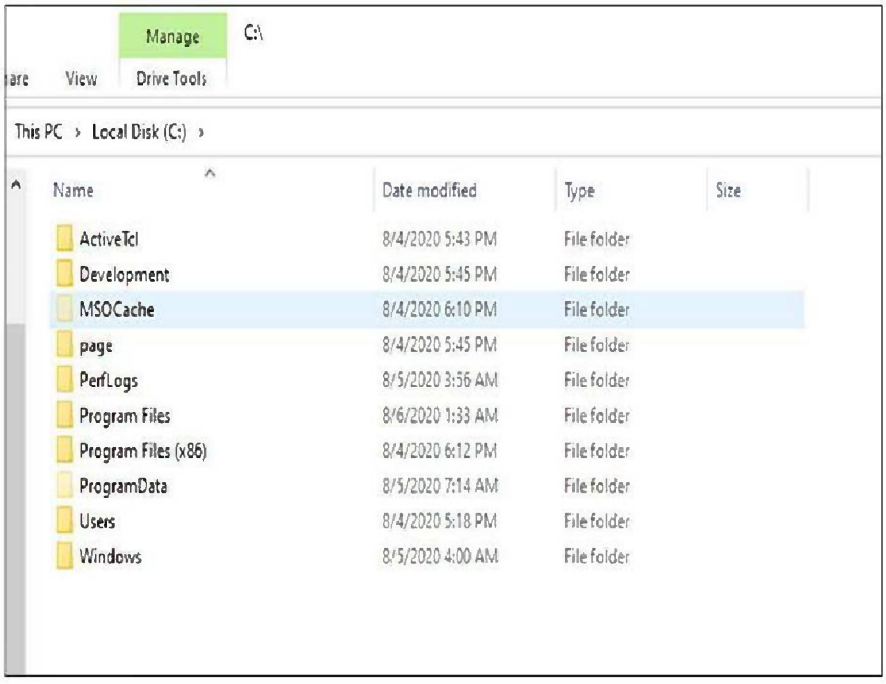


Figure: Windows Directory Tree

Under my Linux machines, I usually keep my Development folder directly under the `/home/greg` folder. Under the Development folder are all my various projects in their own folders.

Under Linux, it is simple to get to my Development folder by using the Nemo (under Linux Mint or Nautilus under Ubuntu) GUI file manager to navigate directly

to the folder that I want to work in, then right click within that folder to open a new terminal. Once I am in the terminal, to get PAGE running, I simply type:

```
$ page
```

Under Windows, it is not quite that easy, but not too much different. Using the Windows GUI File Manager, navigate to the folder you want to work in. Next, type {Ctrl} + L. This will allow you to type within the address bar, which normally shows the path. Now, simply type "`cmd`" and press enter. This will open a new command box. Now, in the new command prompt box, type either:

```
c:\page\winpage.bat
```

or if the PAGE folder has been added to your path, just type "`winpage`".

Getting support

When you are beginning to work with a new program, there will be times that you will run into issues. It might be something you do not understand or maybe a potential bug. Sometimes, it might just be that you have an idea but do not know how to actually implement it in the program.

PAGE offers several options to get some of the answers. Here are just some of the ways:

- Provided documentation: PAGE comes with extensive documentation that is provided with each copy of the documentation. It is located in the PAGE installation folder under the **docs** folder. In this folder, there is an epub version as well as an html version. There is also a 4-chapter tutorial located there.
- Provided examples: Don has provided numerous example programs to help you along on many different subjects. This is located again in the same folder you installed PAGE under the examples folder. If you need examples for Python 2.x, there is the examples-py2 folder.
- Official support: If you cannot find the answer to your questions in the documentation or in the examples, you can always go to the official PAGE Discussion Forum. There is a good chance that someone has already asked the same question that you have. If not, you can always ask your question there. It might take a day or two to get an answer, but you will get an answer. The forum is located at **https://sourceforge.net/p/page/discussion/**.

Last thoughts

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 IDLE with the -i (idle -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, as it is a very simple but powerful editor or VS Code which allows you to debug your code by single stepping through selectable areas. Many programmers prefer other editors like Sublime Text while others like IDLE. The bottom line is to use whatever editor you feel productive using.

Now, you should have gotten PAGE installed on your system and created a working folder to hold all the code you will be creating. So, let us get started!

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 1
Introduction to PAGE

Introduction

It's now time to start learning PAGE. This chapter will concentrate on the basics of using PAGE and creating your own GUI front ends for your programs. Remember, the whole point of PAGE is to allow you to rapidly design and modify graphical interfaces.

The process of creating your GUI is to place widgets, or controls, onto a blank designer form. With PAGE, we can place those widgets by simply clicking on the widget you desire in the Widget Toolbar and clicking on the designer form somewhere close to the position you want it to live. You can move widgets by simply dragging them around the form or using the keyboard arrow keys to "fine-tune" their position.

What you will learn

Following are the topics that you will learn in this topic:

- Starting PAGE within your Operating System
- The major functions each window provides
- Basic use of the Label widget
- Basic use of the Button widget
- Basic use of the Frame widget

- Basic use of the Radiobutton widget
- Creating a GUI form
- Saving a PAGE project

As I said in the Introduction, one important step to a successful project is the way you start PAGE.

Starting page

This section will show the best ways to start PAGE for your operating system and a quick overview of all of the windows that make up the PAGE program.

If you haven't already done so, create a folder to hold your source code and assets (images, databases, and so on.).

If you are running Linux, MacOS, or Raspberry Pi, use the GUI File Manager, navigate to your development folder, and use the right mouse button to get the context menu. Select "`Open in Terminal`". Once the terminal is open, assuming you set up your terminal resource file, type "`page`".

If you are running Windows, use File Explorer to navigate to your project folder. Once there, press *{Ctrl}-L* to edit the address bar and type "`cmd`". This will open a new command prompt and type "`winpage.bat`" or "`c:\page\winpage.bat`".

No matter which operating system you are running, you should be presented with 5 different Windows arranged around your screen. It should look something like the

one shown in *Figure 1.1*:

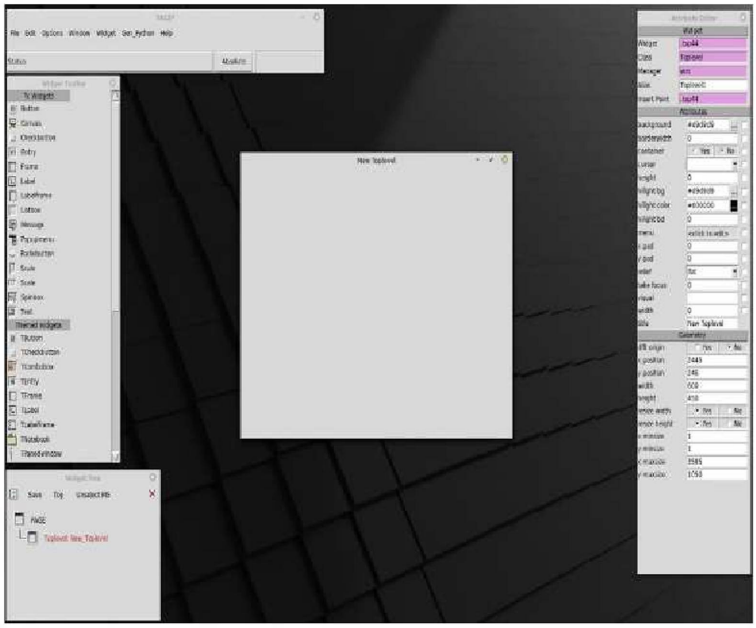


***Figure 1.1**: The Page Windows*

The PAGE designer forms

With 5 different windows taking up the majority of your screen, it's hard to know where to start the first time you run PAGE. Let's take a moment to take a look at each window to see what it is for.

The Main Window

The main window as shown in *Figure 1.2* is important for many reasons. It contains the commands to save your project and generate your Python code. There are many other options that you will occasionally use, but for the most part, you will use this window the least out of all the other windows as shown in *Figure 1.2*:

***Figure 1.2**: The Main Window*

At the very top left of the window, you will find the menu. At the bottom left, you will find the status box, which shows things that PAGE wants you to know. Near the right-hand side, again on the bottom, is a button that will say either "Absolute" or "Relative". You can click on it to change the placement mode. The Absolute mode

will use an exact X/Y position for the placement of your widgets. In the Relative mode, a more general method for the placement is used. The big difference is that if you select the Relative mode when the designed form is resized, the widgets will resize along with the form and the positions of the widgets will move in relationship with the size. In the Absolute mode, the size and position of all the widgets remain the same, no matter how the user resizes the form.

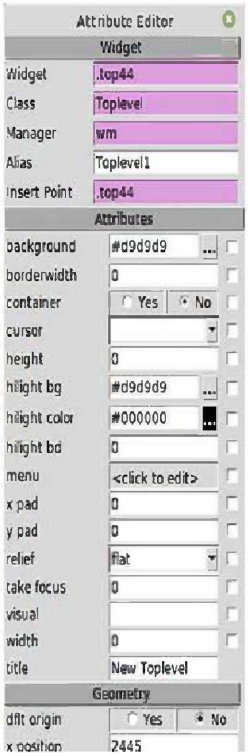
When I design a GUI, 99% of the time I use the Absolute mode, yet Don, the author of PAGE uses the Relative mode all the time.

We'll explore some of the other menu options as we go through the next few chapters.

The Attribute Editor

The Attribute Editor shown in *Figure 1.3* is one of the windows that you will spend a great deal of time using. This is where you set the various attributes of each widget. It is divided into three sections: Widget general attributes and Attributes section is where you can set specific information for things like the text displayed on the widget or a command callback function. The final section is the Geometry section

where you can set the width/height and the X/Y position as well as a few other things:



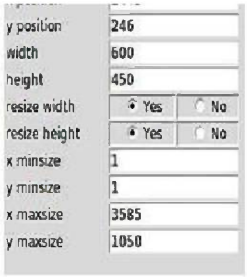


Figure 1.3: The Attribute Editor

This is one of the windows where you will spend a great deal of time using. This is where you set the various attributes of each widget. It is divided into three sections, Widget general attributes, Attributes where you set specific information for things like the text displayed on the widget or a command callback function. The final section is the Geometry section where you set the width/height and the X/Y position as well as a few other things.

If one or more of the sections don't show the proper attributes, simply click on the gray bar for that section and it will expand out so that you can see all the attributes.

The Attributes section changes, depending on the specific widget you have selected. For example, some of the widgets don't have a command callback option or attribute, so it will not show for those widgets.

We'll discuss various widget attributes later in this chapter and in *Chapter 4, Standard Widgets* as well as the various projects as we go forward.

The Widget Tree

Continuing around the screen in a clockwise manner, we get to the widget tree as shown in *Figure 1.4*:

Figure 1.4: The Widget Tree

Every widget that you add to the form will show up here as well as on the main form itself. This provides a quick and easy way to select a widget without the risk of moving the widget, which can be a concern in very complex forms that have several widgets. We'll look at this more throughout the other chapters.

The Widget toolbar

The Widget toolbar shown in *Figure 1.5* will be one of the windows that you will

spend a lot of time with. This scrollable window holds all the widgets that you can use to create your GUI form:

Figure 1.5: The Widget Toolbar

Think of it as an artist's palette of colors. When you want to put a widget on your main form, simply click on the widget to select it (the status area of the main window will acknowledge your selection by changing to something like "Insert Button"), then move the mouse button to the general location you wish to place it on the designer form and click on the form with your left mouse button. That's it. You don't even need to drag and drop it. Click once on the toolbar then click again on the designer form.

I will probably be referring to this window as the "toolbox" from here onwards.

The Designer form

If the toolbox is the artist's palette, the Designer form is the canvas. This is not to be confused with the canvas widget. It is officially called the **TopLevel** widget (Refer to *Figure 1.6*). The designer form is blank when you start PAGE:

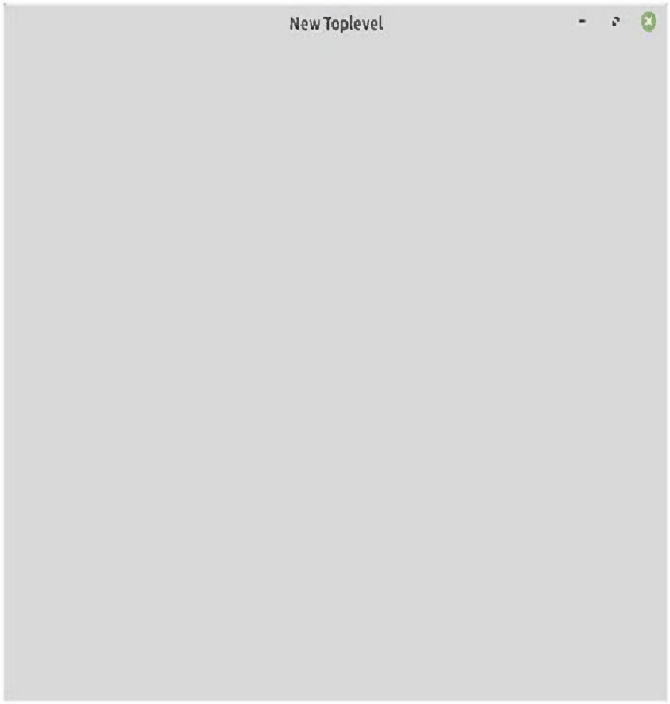


Figure 1.6: The Designer Form

That confuses many people, but that's what it is really called. It is actually a widget, just like a button or static text label. This holds all the other widgets that make up our GUI. At the top of the form is the title. You can modify this using the Attribute Editor or you can leave it with the default and change it in code at any time. We'll see that later on. It also has an alias by default of "Toplevel1". Again, you can change this or leave it with the default.

The Layout of the windows

Since there are so many windows for PAGE and since they are pretty big, the default layout might not work for you. Any or all the windows can be moved and resized at will. To move a window, simply click on the title bar of the window and drag it to where you want. To resize a window, move your mouse cursor to the edge of the window and click and drag to whatever size you wish. Feel free to move things wherever you find to be the most productive layout. To have PAGE, remember the new layout, start a new project in PAGE, and arrange the windows to whatever position and size you want. Then, without adding any widgets, simply close PAGE. When you reopen it, the different windows should all be in the size and position that you left them in. You don't have to manually save anything.

GUI design

Our first project will focus on the basics. We will create a form with multiple widgets on it, and learn how to control them from code.

Here is what our GUI should look like when we are finished:

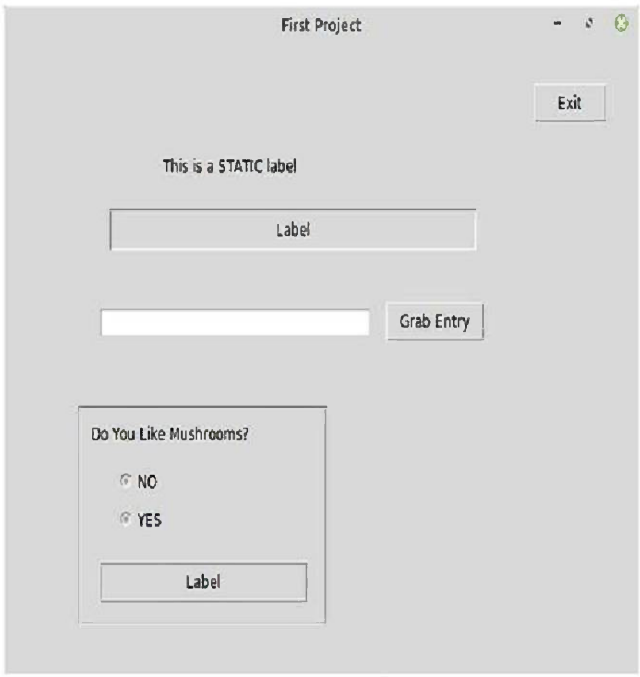


Figure 1.7: Our first GUI form

I won't tell you exactly where to put the widgets. All you need to do is place them somewhere on the main form, making your form look something like the one in *Figure 1.7*. Let's get started.

The first thing to put on the form is the Exit button. Using the toolbox, select the button widget, which should be the top icon as shown in *Figure 1.8*.

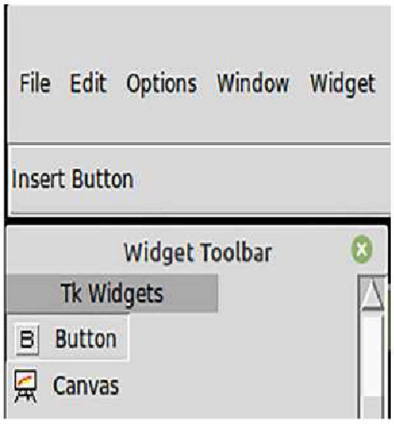


Figure 1.8: Button icon in Toolbox

If you don't see it, use the scroll bar on the right of the toolbox and scroll all the way to the top. Click on the **Button** icon and you should see the '**Insert Button**' appear in the status box of the main PAGE window. It should go somewhere on the upper right-hand corner of the form. Move your mouse cursor to somewhere close to the right side of the designer form. To place the button, simply click on the mouse Button1.

> ***Note: Tk and Tkinter refer to the buttons of the mouse as Button1, Button2, and Button3. When you look at the mouse, the left button is normally Button1, the center button is Button2, and the right button is Button3. You can, of course, reassign the buttons in the OS if you are left-handed.***

Notice the eight black boxes along the edges of the Button widget as shown in *Figure 1.9*. These are the resize handles for this widget:

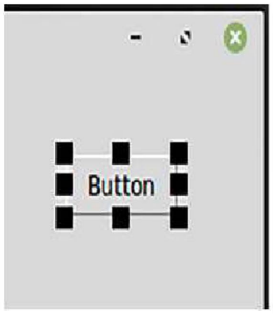


Figure 1.9: Button on Form

To move the widget, you can hold down the left mouse button in the center of the widget and drag it wherever you what it. To resize the widget, click and hold the left

mouse button on one of the resize handles and drag your mouse until it's the size you want.

Once you have it in the general area you want it, take a look at the Attributes Editor:

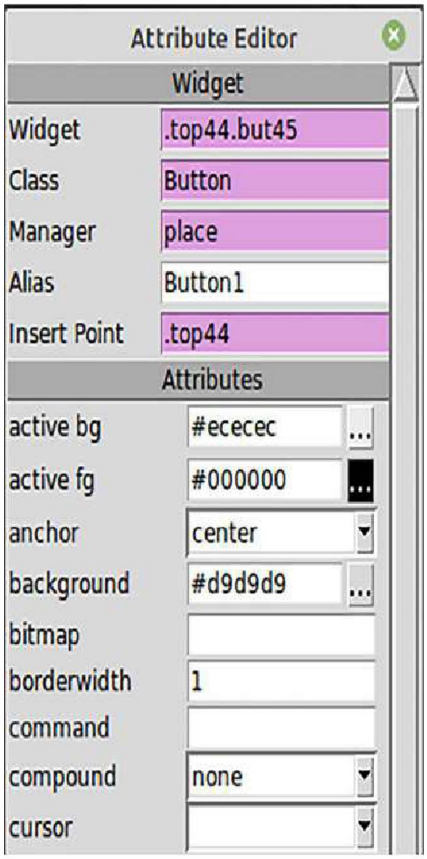


Figure 1.10: Attribute Editor for Button

Notice that the preceding image only shows a part of the **Attribute Editor**. The first thing you should see is that the **Alias** is set to **Button1**. It is located near the top of the window under the **Widget** section. Alias is a "friendly" name for this widget. You can call it whatever you want, but we will use a name that explains what that widget does. Use your mouse and select the text "Button1" and change it to "**btnExit**". That way, we know that the widget is a button and its purpose is to allow the user to click on it which causes the program to exit (once we add a little bit of code).

Now, we need to set the various attributes for this widget. We've already set the Alias which is in the Widget section. There are two more attributes that we need to set, which are located under the Attributes section of the Attributes Editor. The first is the command attribute. This is where we set up the Callback function. Many widgets have this attribute, but not all of them. A callback is a function that will be "called" when the user clicks on the widget or changes something about the widget.

For our Button, it is just when the user clicks on it. We simply provide the name of the function and PAGE will create a skeleton of the function for us. The final attribute we will set is the text that the button will display. In this case, we want it to show "Exit". When we need to set attributes, we'll put everything in a grid with two columns as shown below. The left column is the attribute name that will need to be changed and the right column is the value that we need to assign to it:

Attribute	Value
Alias	btnExit
command	on_btnExit
text	Exit

This should make it very easy for you to know what attributes and values to use as we go through the various projects in the book.

Now that you have set the attributes for our first widget, let's save the project, just to be safe. In the main PAGE window (which should be in the upper left-hand corner of your screen), select **File** and then in the drop-down menu, select **Save**. You will see a dialog box coming up asking you what to call the project:

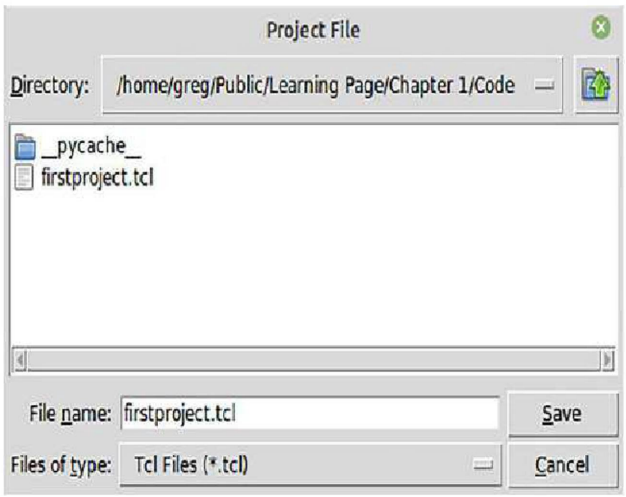


Figure 1.11: File Save Dialog

Let us save as "**firstproject.tcl**". At this point, we can use any filename we like but do not use dashes or underscore characters and use a .tcl extension. PAGE should automatically add the .tcl extension. Click on the **Save** button.

Now, we can continue adding widgets. The next things we will add are two Labels. Labels are good for user prompts that sit next to another widget, like an Entry widget, to tell the user what information that widget is expecting, or to draw attention to a group of widgets without using a frame:

***Figure 1.12:** Label in Toolbox*

These types of Labels are considered Static labels since their text will not change during the run of the program. You can, however, make the Label dynamic, where the text can change during the run of the program. The first Label will be a static type and the second will be the dynamic type. We will use the same base widget for both types. In the Toolbox, find the Label icon (the sixth icon from the top), left click on it, and then click within the main Design form to place it. Let us place it around **120** pixels in (X-axis) and **80** pixels from the top of the form (Y-axis). You can watch the X, Y, Width, and Height values by scrolling through the Attributes Editor to view the **Geometry** section. These values will change as you move or change the width or height. You should drag the right side of the label (the center black handle on the right) to make it longer. Let us make it about **190** pixels wide. Let us also keep the **anchor** attribute set to the default value **Center**. We will see more about this attribute later on.

The following grid lists the attributes you should set in the Attributes editor for this first Label:

Attribute	Value
Alias	lblStatic
text	This is a STATIC label

You now need to add another Label slightly below this. This will be our dynamic label. Let us put the demo at **X: 100** and **Y: 120** and set the width to about **350**. The reason to make this so wide is that this is dynamic by nature and we don't know what the text will be. Again, we will leave the **anchor** attribute set to the default **Center**:

Attribute	Value
Alias	lblDynamic
text	Label
relief	Sunken
text var	DynamicLabelText

We'll pause here for a moment to learn about the last two attributes. The **relief** attribute allows you to change the look of the widget by including a border around the widget. The possible values for this show up in a dropdown when you click in the box are **flat, groove, raised, ridge and sunken**. You can select each option and

see the result of that option on the widget. Some options look like another option, but that can change by modifying the **border width** option. We'll look at that later in the book.

The **textvar** attribute option shown in *Figure 1.13* is what allows the label to change from static to dynamic:

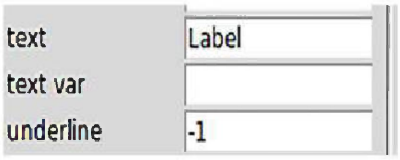


Figure 1.13: Text var attribute

When you change this attribute, the value you type in the box will become a special type of string variable that you can reference in your code to set or get the current value of the text of the label. We'll see this more in detail when we write the code but know that PAGE will set this up for you when you generate the Python code.

Next, we will need an **Entry** widget shown in *Figure 1.14*. This widget allows the user to type in a single line of text:

Figure 1.14: Entry Widget in Toolbox

This data can be read on demand via code. Let us place our widget **at X: 90, Y: 190, Height: 21**, and **Width: 256**. Use the same method that we used on the button and the labels to select the Entry widget and place it on the main design form.

The attributes you need to set for this widget are shown below:

Attribute	Value
text var	EntryData

Now, we will add another **Button** widget. This will be used to get the text that the user entered into the Entry widget and display it in the dynamic Label:

Attribute	Value
Alias	btnGrabEntry
command	on_btnGrabEntry
text	Grab Entry

Next, we will place a **Frame** into our form as shown in *Figure 1.15*. The Frame widget is often used to hold various widgets that should be grouped together:

Figure 1.15: Frame in Toolbox

In our case, we will be putting a static Label as a prompt, two Radiobuttons that will act as easy Yes or No indicators and another Label that will dynamically show the Radiobutton selection. We won't change any of the Frame's attributes for this project.

Imagine that there are four equal sections within the Frame. Each section runs horizontally. Now, place a Label in the top imaginary section, which will be our static Label:

Attribute	Value
Alias	lblRbQuestion
text	Do You Like Mushrooms?

We are almost done with our first project. We will put the **Radiobuttons** inside the frame as shown in *Figure 1.16*.

Figure 1.16: Radiobutton

Again, thinking of the four imaginary sections, the Radiobuttons will go into sections 2 and 3.

Attribute	Value
Alias	RadiobuttonNo
command	on_rbClick
text	NO
value	0
variable	selectedButton

Notice here that we have some extra attributes. One of these is already set for us by PAGE, which is the **`variable`** attribute. Radiobuttons act as one-of-many quick selection widgets, so we will set the attributes for command and variable to the same thing for all of our Radiobuttons. The alias, text, and value are unique within the separate widgets. The **value** attribute is an integer that is returned in the **`selectedButton`** variable:

Attribute	Value
Alias	RadiobuttonYes
command	on_rbClick
text	YES
value	1
variable	selectedButton

Finally, we will add another dynamic label that will display what the user has selected. While the Radiobuttons are fairly self-explanatory, this has been done to show the automatic callback capabilities:

Attribute	Value
Alias	lblRbResponse
relief	Ridge
borderwidth	2
textvar	rbSelectDisplay

Now that our GUI is all designed, take a moment to rearrange the widgets to provide the most pleasing visual layout that you can find. My idea of an ideal GUI form might not be the same as yours.

Let's save the code once again and generate the Python code.

On the Main window, select **`File | Save`**. Since we've already saved the project earlier, we won't get the dialog box this time (unless you select **`Save As`**). To the right of this is the **`Gen_Python`** menu item. There are four options under this item. They are as follows:

- Generate Python GUI
- Generate Support Module
- Load Page Consoles
- Load Project Into IDE

At this point, the only options we need to deal with are the Generate Python GUI and Generate Support Module. Be sure to select them in the order that is being shown.

First, click on the Generate Python GUI item. You will see a dialog box similar to the one shown in *Figure 1.17*:

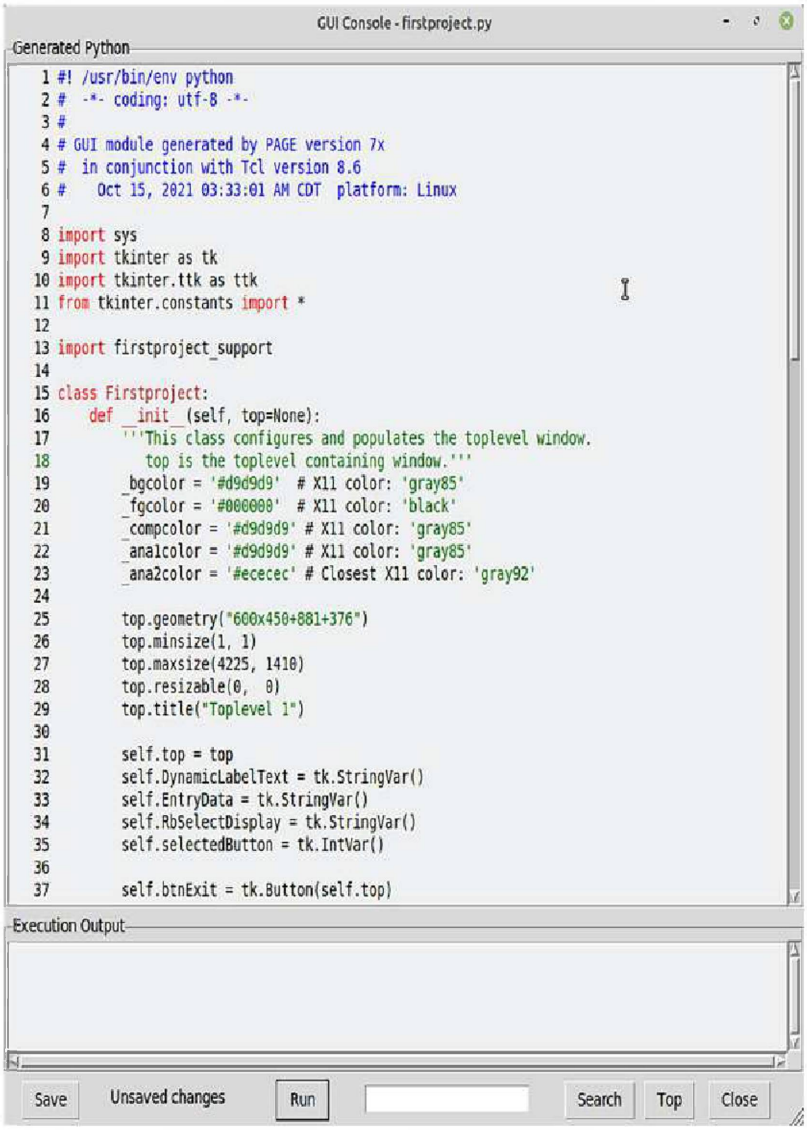


Figure 1.17: GUI Generator Dialog

You can see the generated code in an editor in the top part of the dialog box. At the bottom, there are several buttons. For us, right now, the far left button marked Save is the important one. You don't need to change any of the code for the GUI module. Click on Save and the dialog box will go away. The file is saved as, in our case, firstproject.py.

Now, click on **Gen_Python** again and select Generate Support Module. *Figure 1.18* shows what the support console looks like:

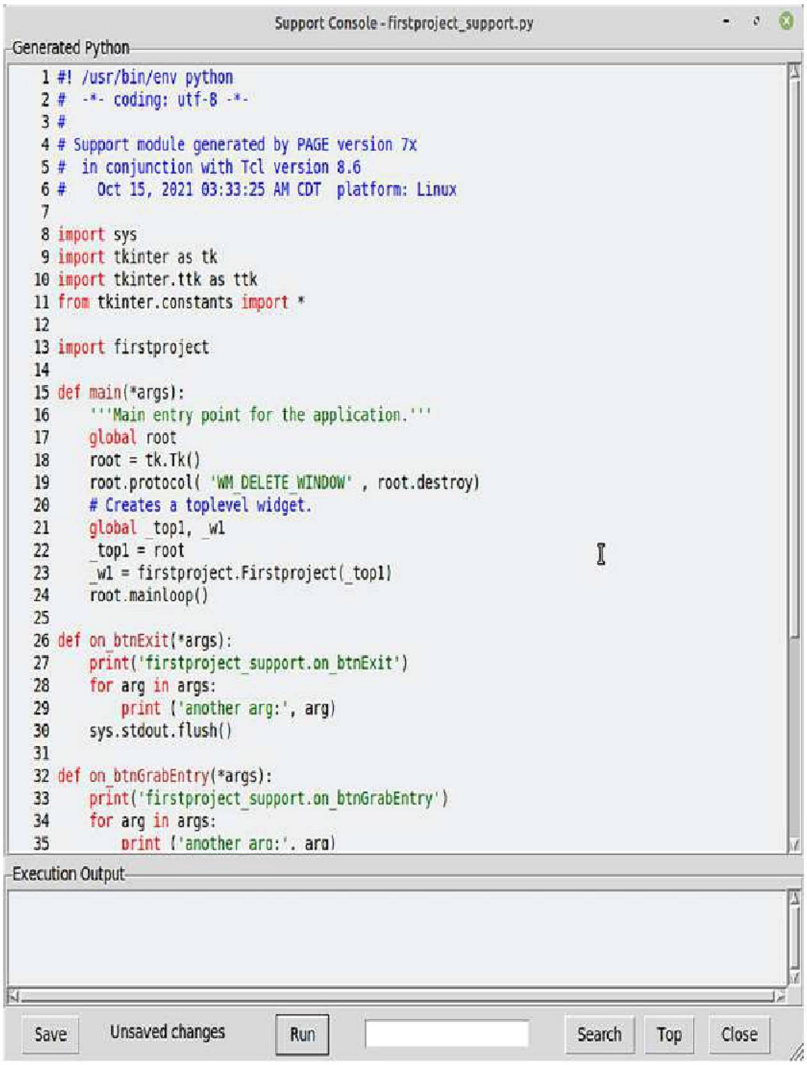


Figure 1.18: Support Generation Dialog

If you need to add any code, you CAN do it here, but at this time, just click on the Save button. This will generate the `firstproject_support.py`

Now that everything is done, you can close PAGE and move to your IDE or text editor. Remember, we have created three files. The first is the PAGE .tcl file that provides all the information about our project so that we can use PAGE to update the GUI if needed. The other two are our Python files. The first is simply, in this case, named the same thing as our Tcl project file and the other one is the project name with "_support.py" added. The first Python file is known as the GUI file and should NOT be edited. The second file, you might have guessed, is called the Support file. This is the file that you want to edit to hold all the code that glues the widgets together into a comprehensive application. The reason that you should not edit the

GUI file is that if you ever need to edit the GUI part of the project in PAGE, when you save and re-generate the Python files, the GUI Python file will be completely rewritten. The Support file generation process will ask you if you want to update the Support file (normally you will answer Yes) so that you can keep your code.

The Code

Since this is such a short program, we'll go through the complete code in the support module. Many sections of code are presented here just for discussion purposes and we won't need to modify them.

At the top of the file are some comments and the "standard" shebang line. Next, comes the library import section. Again, you don't need to change this part of the code but you need to be aware of it:

```
import sys

import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *


import firstproject
```

In later projects, you will need to import extra modules that PAGE doesn't know about, so you would need to edit this section. Usually, we put any extra imports just after the import sys line so that they are all together.

Remember when we were designing the GUI form, we set some text variables. In earlier versions of PAGE, you could see a function in **the _support** file. However, in version 7, the variables we set are located in the GUI file. Here is what the definitions for the variables look like:

```
        self.top = top

        self.DynamicLabelText = tk.StringVar()

        self.EntryData = tk.StringVar()

        self.RbSelectDisplay = tk.StringVar()

        self.selectedButton = tk.IntVar()
```

Again, you don't need to change these you should NEVER modify the GUI file because if you ever change or update the layout in PAGE, you will lose all your changes, since the GUI file is rewritten from scratch every time you generate it.

The first full function in the support file is named main. Here is what it looks like for our first program:

```
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 = firstproject.Firstproject(_top1)
    root.mainloop()
```

It's important to understand that this is the very last function that runs before the program starts and your GUI is shown to the user. If you need to do anything like setting variables or loading a database, you need to add a line into the main function to call it. We need to set some things before the program runs, so we need to add a line to call the initialization function. We'll call that function init. We'll place our new line in between the last two lines of the main function:

```
    _w1 = firstproject.Firstproject(_top1)
    init()
    root.mainloop()
```

Also notice the two lines in the main function that say

```
    global _top1, _w1
    _top1 = root
    _w1 = firstproject.Firstproject(_top1)
```

Whenever you want to refer to a widget on your form or a tkVariable that you have defined, you will need to preface it with **_w1**. This allows Python to know that it is a widget or variable defined in the GUI module.

Next, we need to create our init function. We'll place it just after the main function:

```
def init():

    _w1.DynamicLabelText.set("Hi!  I am a dynamic Label")

    _w1.selectedButton.set(0)

    _w1.RbSelectDisplay.set("")
```

Again, we preface each of the variables with the _w1 reference. In each of the lines, we are using the .set method to modify the widget attributes. In the first line, we change the text in the dynamic label to "Hi! I am a dynamic Label." The second line sets the first Radiobutton that says "NO" to be selected by using the value that we defined in the PAGE designer. Finally, we clear the text from the label that will display which Radiobutton you or the user clicked on. Since we are just starting the program, there is nothing to display.

Now, we'll deal with our first callback function. This function gets called whenever the user clicks on the Grab Entry button and pulls the information the user has typed into the Entry widget. When we defined the command method for this button, PAGE created the function with just the first two lines. We are going to add two more lines below them. The two lines that need to be added are shown in bold:

```
def on_btnGrabEntry(*args):

    print('firstproject_support.on_btnGrabEntry')

    for arg in args:

        print('another arg:', arg)

    sys.stdout.flush()

    Entered = _w1.EntryData.get()

    _w1.DynamicLabelText.set(Entered)
```

So, to retrieve the data from the Entry widget, we use the **.get()** That is put into the Entered variable. Then, we show that data into the dynamic label with the **.set()**

Next, we need to fill out the callback for the Exit button. PAGE creates a function for us that properly ends our program called **destroy()**. All we have to do is call that

function from this callback function. Since PAGE created the skeleton for us when

we assigned the command attribute of the button, all we need to do is make the call to the `root.destroy()` method:

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

The last thing we need to edit is the callback function that gets run when the user clicks on one of the Radiobuttons. In this function, we will figure out which Radiobutton was clicked and then put some user-friendly text into the dynamic Label in the frame:

```
def on_rbClick(*args):
    print('firstproject_support.on_rbClick')
    for arg in args:
        print('another arg:', arg)
    sys.stdout.flush()
    WhichSelected = _w1.selectedButton.get()
    if WhichSelected == 0:
        _w1.RbSelectDisplay.set('You picked "NO"')
    else:
        _w1.RbSelectDisplay.set('You picked "YES"')
```

The first thing we do is call the `.get()` method of the selectedButton variable. Remember that we associated a number with each of the Radiobuttons by defining it in the value attribute. The NO Radiobutton returns a 0 and the YES Radiobutton returns a 1. So we check this value and put the appropriate text into the dynamic label.

And finally, the entry point for our program if the user runs the support module instead of the GUI module:

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


Finally, we can run our program by using the command line:

```
$ python firstproject.py
```

You will see something like this:

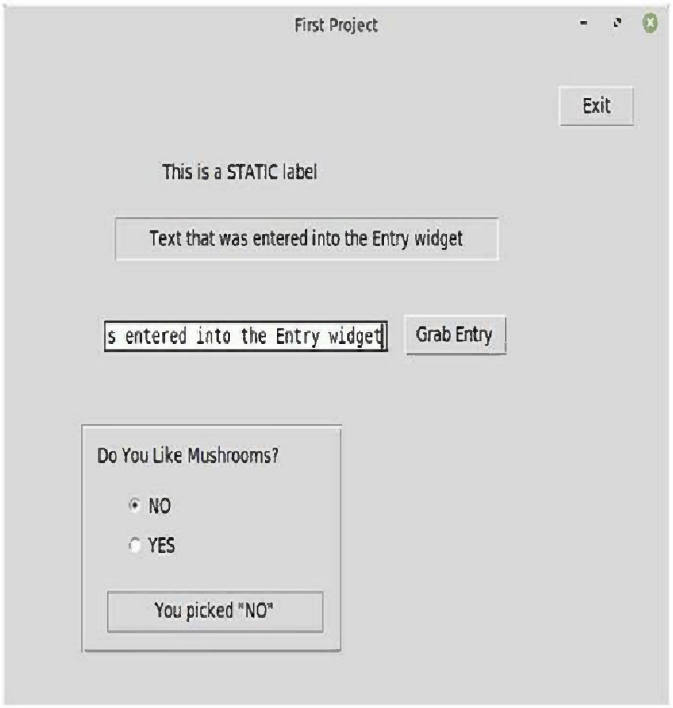


Figure 1.19: Finished Application

Let us enter some text into the Entry widget, click on the Grab Entry button, and then click on the NO Radiobutton to show what you should expect.

Conclusion

Congratulations! You've created your first PAGE GUI program. Even though you might have never used PAGE before, you now can see how easy it is. Things will get easier for you from here. Once some users try PAGE, they never go back to non-GUI programming. Things are so easy to do. You have already learned several things such as:

- How to start PAGE correctly
- How to place various widgets on the form
- How to generate the two Python modules

- How to generate the two Python modules
- How to set attributes to create a dynamic Label

- How to get the data that a user has entered into an Entry widget
- How to properly close and exit an application.

In the next chapter, we will look at creating a program that has multiple forms and how to create a program that has a splash screen.

Multiple Choice Questions

1. How many files does PAGE produce for any given project, not counting backup files?
   a. One
   b. Two
   c. Three
   d. Four
2. Of the following files, which should you NEVER edit by hand?
   a. {projectname}.tcl
   b. {projectname}.py
   c. {projectname_support}.py
   d. Both a and b
   e. Both b and c
   f. None of the above
3. The attribute that allows a static label to become a dynamic label is _____.
   a. Alias
   b. text
   c. text var
   d. None of the above

Answers

1. C – PAGE produces three files for a project, not including .bak? files. One is the .tcl file that PAGE uses directly and two Python files, a {projectname}.py file, which is the GUI file and a {projectname_support}.py file which is the support module.
2. D – The {projectname}.tcl file and the {projectname}.py (GUI) files should NEVER be edited by hand.
3. C – The text var attribute allows the displayed text of a label to be changed dynamically from the code.

Questions

1. What would be the advantages of a GUI program over a Command Line

1. What would be the advantages of a GUI program over a Command Line interface program?
2. What would the disadvantages be of a GUI program over a Command Line interface program?

Answers to the questions

1. GUI programs are easier for a user to deal with when there are multiple inputs that the user might need to deal with. In addition, the GUI program responds to events, rather than just using sleep() functions. This allows the program to do other things while waiting for an input from the user.
2. The biggest disadvantage of a GUI over a CLI program is when changes are needed to be made in the flow of inputs and outputs of the program. In a CLI program, you can just put the changes in the middle of the other code. In a GUI program, the form(s) would, at the very least, need to be re-worked everytime a change is requested from the end client.

Key Terms

- Attributes
- Widgets
- Static Label
- Dynamic Label

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2
Going Further

Introduction

Since you now know the basics of using PAGE, it's time to look at creating more advanced programs. In this chapter, we will learn how to create PAGE programs that have two (or more) forms and how to create applications that have a splash screen.

What you will learn

In this chapter you will learn the following topics:

- Creating a program that has two forms
- How to share data between the two forms

- How to limit multiple instances of the second form
- How to start PAGE to edit an existing project from the command line
- How to change fonts for widgets
- How to change colors for widgets

Section 1 – Multiple forms

When new PAGE users move past the very basics, one of their first questions is "How do I create an application with multiple forms?" This section will show you just how easy it is to create an application that uses two forms. Once you have the basics, the number of forms that your application can have is only limited by your imagination.

Designing the GUI

As I said, this project will have two forms: a parent and a child form. This is what the parent should look like when we get through with the design:

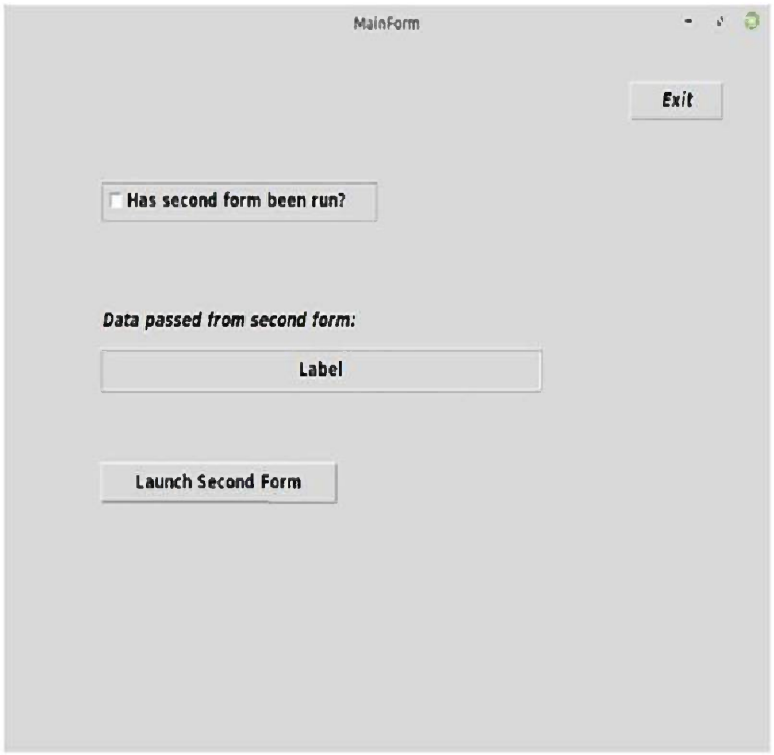


Figure 2.1: Parent Form

You will notice that we will have a new type of widget on this form, a Checkbutton widget. This is very useful for Yes/No type options or to simply show that something has happened or not. The button will launch the child form near the bottom of the form.

The child form should look something like this:

Figure 2.2: Child Form

The child form looks different from our parent form and the form we created in the last chapter. We also have an Entry widget that the user can type in. When the user clicks on the Exit button on the child form, that information is passed back to the parent form for display. The form background and many of the widgets are of different colors.

Enough of talking, now let's get started. Create a new folder for our project named **DualForms** and start PAGE. As with any project, we'll start with the main form first. Make sure that the various windows are in convenient places and expand the size of the form a little bit. Try to move it somewhere so it is centered on the screen. Don't put too much effort into centering the form right now, since we'll write some code to do this automatically on startup. I made mine 764 pixels wide and 506 pixels high. The very next thing you should do is to alias the top-level form itself. In this case, we'll call it **MainForm**. You can also set the title at this time to **MainForm** as well:

Figure 2.3: Setting the top-level Alias

As we did in the last project, place a **Button** widget somewhere in the upper right-hand corner of the form for our Exit button. This time, we'll also set the font for our Exit button. In order to do this, find the font attribute in the Attribute Editor and click on the button with the three dots. This will open the font editor dialog:

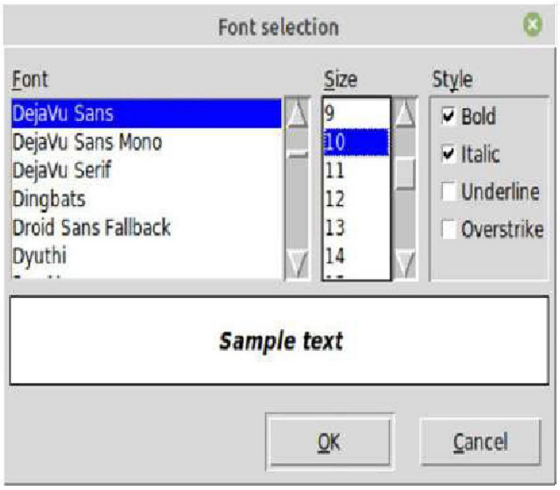


Figure 2.4: Font selection dialog box

Now, go ahead and set all the attributes as we have before:

Attribute	Value
Alias	btnExit
command	on_btnExit
text	Exit
font	DejaVu Sans Bold & Italic

If you find that you don't have a font named **DejaVu Sans**, pick a font that you like and check the Bold and Italic checkbuttons.

Next, we will add a **Checkbutton** widget. I placed mine at X: 90 and Y: 100 and made the height 31 and the width 263. I made it so wide that the text would show and not possibly get cut off:

Figure 2.5: Checkbutton

Attribute	Value
anchor	w
border width	2
relief	ridge
text	Has the second form been run?
Variable	Che46

Now, we'll add a static **Label**. I put mine at X: 90 and Y: 190 and I made it pretty wide, about 323 pixels. This will just be a prompt to show what the next widget is there for. I left the alias with the default value (Label1).

Attribute	Value
anchor	w
text	Data passed from the second form:

And right below it, we'll put another **Label**, but this one will be dynamic. The location that I put mine was X: 90 and Y: 220. I made this one a bit wider, about 420 pixels since I don't know what the user will be sending back from the child form:

Attribute	Value
borderwidth	2
relief	ridge
textvar	SecondFormData

Finally, we will add another **Button** somewhere near the other widgets. When you click on them, it will launch the second form:

Attribute	Value
Alias	btnLaunch
command	on_btnLaunch
text	Launch Second Form

Now save the form as `DualForm.tcl` and generate the two Python modules. Once you've done this, we can create the child form.

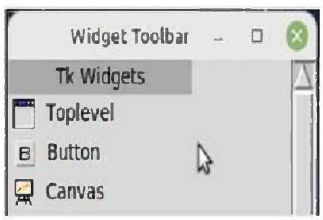


Figure 2.6: Toplevel button in the Toolbar

In earlier versions of PAGE, we would have to start an entirely new project for our child form. However, with PAGE 7, all we have to do is click on the Toplevel button in the Toolbar.

When you click on the **Toplevel** icon in the Toolbar, a new form will appear. It will appear close to the first form. In the Attributes editor, we need to set the alias and the title. Set the Alias to **ChildForm** and the title to **Second Form.** Move the form to cover the **MainForm** and resize it a little bit. I set mine to 835 pixels wide and 439 pixels high.

Sometimes, you will want your child forms to look differently from the main form so it's easy to tell the difference. We'll do this by setting the background color of the child form. As I stated earlier, we'll be setting the colors for almost all the widgets in this form. Take a look at the Attributes Editor:

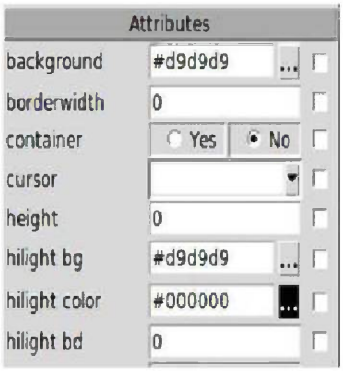


Figure 2.7: Background color attribute

This image is just a section of the Attributes Editor that deals with the color attributes of the TopLevel designer form. Some widgets will have more, while some will have less. If you look closely at the image, you'll see that there is a small button just to the right of the color attributes. If you click on that, you will see a dialog box pop up that looks something like this:

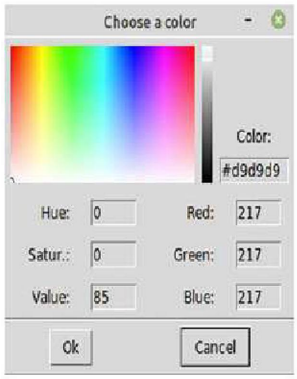


Figure 2.8: Color dialog

There are four ways that you can set a color attribute for any given widget (including the TopMost form widget) using this dialog. You can set them by:

- Clicking on a color in the pallet box, tweaking with the side white to the gray slider
- Entering the color HEX values
- Setting the Hue, Saturation, and Value entries
- Setting the Red, Green, and Blue values

Going into a proper discussion about this would probably take up an entire book in itself. There are a ton of websites that explain these processes much better than I could. There IS a fifth way, which is much easier, than dealing with the dialog as long as you are willing to use one of the pre-defined Tkinter colors. You can simply type in the standardized color name that Tk and Tkinter use.

Take a look at the following website:

http://www.science.smith.edu/dftwiki/index.php/Color_Charts_for_TKinter

It contains a chart by color along with the name of the color that you can type into the attribute box. I threw together a program that does the same thing and here is a screenshot:

Figure 2.9: Tkinter Named color chart

For the most part, we will only be changing a color or two for each widget, although some widgets have many possible color attributes.

So, to set the background color for the form itself, simply type "cornflower blue" in the background color attributes box:

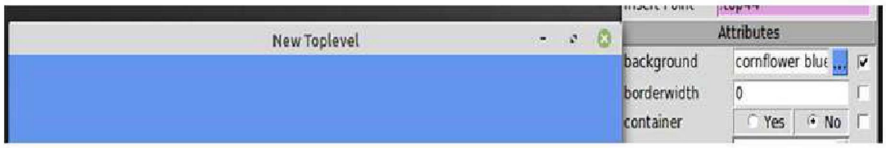


Figure 2.10: Setting the background color to cornflower blue

Yes, you can include spaces in the color name.

Now, we'll add a Dismiss button just like we have done in all our projects so far. As usual, place it somewhere near the top right-hand corner of the form. This button will hide the Child form and show the Main form again:

Attribute	Value
Alias	btnDismiss
command	on_btnDismiss
text	Dismiss
font	DejaVu Sans Bold 10 pt Italics

Next, we need to place four labels on our form. This time we'll just leave the Alias to the default. Place them so that your form looks close to the following image:

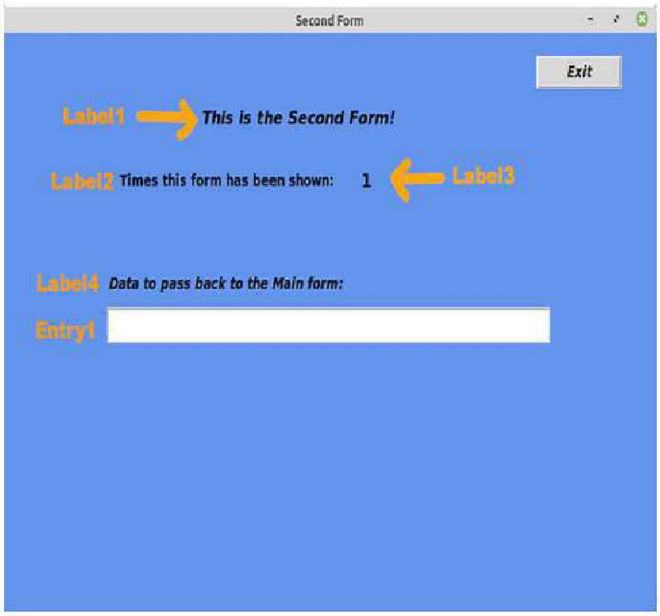


Figure 2.11: Second form widgets

Here are the attributes for Label1. Notice that all our Label widgets will have the same background color as the form, even the dynamic one:

Attribute	Value
activebackground	Cornflower blue
background	Cornflower blue
font	DejaVu Sans Bold 12 pt Italics
text	This is the Second Form!

You can see that the attributes for Label2 are very close to the same but with a couple of small differences:

Attribute	Value
activebackground	Cornflower blue
background	Cornflower blue
font	DejaVu Sans Bold 10 pt
text	Times this form has been shown

Label3 is a dynamic label, so we'll need to set the textvar rather than the text attribute:

Attribute	Value
activebackground	Cornflower blue
background	Cornflower blue
font	DejaVu Sans
	Bold 12 pt
textvariable	RunCount

Label4 is back to a standard static label:

Attribute	Value
activebackground	Cornflower blue
background	Cornflower blue
font	DejaVu Sans Bold 10 pt Italics
text	Data to pass back to the Main Form

And finally, we need to place an Entry widget to type something to send back to the main form. For this widget, we will keep all of the defaults except to define a

the main form. For this widget, we will keep all of the defaults except to define a

textvariable that can be used to access the typed information. As you can see in the following table, I set it to be pretty wide to allow for more room to type:

Attribute	Value
textvar	PassBack

That's pretty much it for the child form. There is just one more thing to do before we start modifying the code. You need to create an empty file named shared.py. Now, I know this seems a bit odd, but we will use this file to pass information back and forth between the two forms. It is important it is empty and that it is named **`shared.py`**

Now, we can concentrate on adding code to the `two_support` files.

The code

At this point, you should have seven files in your project folder. Three for the main form, three for the child form, and the empty file shared.py. As discussed in *Chapter 1, Introduction to PAGE*, we will work with the _support files. So, let's start with the MainForm_support.py file first.

We need to add two lines to the import section near the top of the file. Again, I have set the two lines in bold that you need to add:

```
import sys

import shared

import ChildForm
```

Be sure that you spell them correctly. We are importing the **`shared.py`** and the **`ChildForm.py`** files, but we don't need to add the **`.py`** extension. Just for safety's sake, verify the spellings with the files in File Explorer or your terminal. This time, we don't need to import the **`ChildForm_support.py`** file.

The next function is the **`set_Tk_var`** function. This holds the variables that we defined during the design process. You don't need to change anything here; I just included it for clarity:

```
def set_Tk_var():

    global che46

    che46 = tk.IntVar()

    global SecondFormData
```

```
    SecondFormData = tk.StringVar()

    SecondFormData.set('Label')
```

Now, we get to the **init** function. There are only two lines you need to add here. One calls the startup function and the last line will center the form on the screen:

```
def init(top, gui, *args, **kwargs):

    global w, top_level, root

    w = gui

    top_level = top

    root = top

    startup()

    centre_screen(723, 506)
```

The startup function takes care of initializing all the variables that we need for this program. Of course, we could have just put it into the init function, but doing it this way, keeps the code clearer. First, we set a variable in the shared "module" called childrunning to False. Notice we preface the variable with "shared". This way, the Child form can access it as well. Then, we set the number of times the child has been run since the start of the program and finally, we set passback to an empty string. We do this because if another program, in this case, the child form, tries to read them, they are already initialized. If we don't do this and the variable hasn't been written yet, the second program will throw an error and crash:

```
def startup():

    shared.childrunning = False

    shared.childruncount = 0

    shared.passback = ''   # two single quotes (')
```

Here is our Exit button callback routine, as discussed in the previous chapter:

```
def on_btnExit():

    print('MainForm_support.on_btnExit')

    sys.stdout.flush()

    destroy_window()
```

This is the "magic" function that will start our child form. There are really only two lines that we need to create since, like any other callback that we set up in PAGE, it has already been created for us. If you look at the code in **ChildForm.py**, you will find two functions:

```
def vp_start_gui():

    '''Starting point when module is the main routine.'''

    ...


def create_ChildForm(rt, *args, **kwargs):

    '''Starting point when module is imported by another module.

       Correct form of call: 'create_ChildForm(root, *args, **kwargs)'
       .'''

    ...
```

Both of these functions basically do the same thing, which is to act as an entry point for the child program. However, it is HOW the function is called that makes the difference. If you look at the **create_ChildForm** function, you will see that the comment just after the definition says that this is the starting point when the module is imported by another module. That's our trick. The function takes one required parameter and an almost unlimited number of optional ones. The required parameter is the root window of the calling program, which is "root":

```
def on_btnLaunch():

    print('MainForm_support.on_btnLaunch')

    sys.stdout.flush()

    ChildForm.create_ChildForm(root)

    hide_me()
```

After we launch the child program, we then hide the main form by iconizing it. The **hide_me** function is a little further along in the code.

The **centre_screen** function figures out how wide and high the screen is and then divides those values by 2, and then subtracts the width and height of the form (also divided by two. Finally, we call the **root.geometry** function to make the form show up there. It requires two parameters, the width, and height of the form itself. We get

that by, once again, sneaking a peek at the class **MainForm** in the **MainForm.py** file. It's set by PAGE and can be found in the line:

```
top.geometry("723x506+587+247")
```

The first four sets of numbers are as follows:

- Form Width
- Form Height
- X (right/left position of the form)
- Y (up/down position of the form)

We want the Width and Height values. We send that information in when we call the function from the **init** function:

```
def centre_screen(wid, hei):

    ws = root.winfo_screenwidth()

    hs = root.winfo_screenheight()

    x = (ws/2) - (wid/2)

    y = (hs/2) - (hei/2)

    root.geometry('%dx%d+%d+%d' % (wid, hei, x, y))
```

Now we have two more "magic" functions. **Show_me** and **hide_me**. The hide_me function simply uses the Tkinter method "**root.withdraw**" which iconizes the form basically hiding it. The **show_me** function is a bit more complicated since we want to make sure that we get some information to and from our shared module.

```
def show_me():

    global root

    if shared.childruncount > 0:

        che46.set(1)

    SecondFormData.set(shared.passback)

    root.deiconify()


def hide_me():

    global root

    root.withdraw()
```

That's it. Save your **MainForm_support.py** file and we can move on. Next, we'll modify the **ChildForm_support.py** file.

The import section looks pretty much like the one for **MainForm**. But with just one small difference. Not only do we import **MainForm (.py)**, but we need to use a function that is inside the **MainForm_support (.py)** file, so we have to import that as well:

```
import sys

import shared

import MainForm

import MainForm_support
```

Again, PAGE generates this function for us, but it's useful to have a list of all the special Tk variables. It's just here for your reference:

```
def set_Tk_var():

    global PassBack

    PassBack = tk.StringVar()

    global RunCount

    RunCount = tk.StringVar()

    RunCount.set('Label')
```

Again, we come to the **init** function; this time for the Child form. It looks pretty much exactly like the one for **MainForm_support**. That's done on purpose:

```
def init(top, gui, *args, **kwargs):

    global w, top_level, root

    w = gui

    top_level = top

    root = top

    startup()

    centre_screen(760, 506)
```

Our startup function is much different from the one for MainForm, but I think you can, by now, figure out what we are doing:

```
def startup():

    shared.childrunning = True

    shared.childruncount += 1

    print(f'ChildRunCount={shared.childruncount}')

    shared.passback = ''   # Two single quotes (')

    RunCount.set(shared.childruncount)
```

The one thing that you might not be familiar with is the center line that starts with "`print(f'…`". The f print function is new since around Python 3.6. However, it is very useful for debugging and creating strings on the fly. Basically, the "f" at the front of the string, as far as I'm concerned, is short for "create a formatted string". Then, the string is laid out as you would normally want it to look, but wherever you want to put a variable inside of the string, you put the variable name within "squiggly brackets". So, if the child program has been run 14 times since the program started, the line would read:

```
ChildRunCount = 14
```

Cool huh? Anyway, we can move on from here.

Again, we have the exit button to deal with. AND it's mostly similar to what we've done before for the Exit button, with a couple of small exceptions.

When we get ready to delete the form, we set the passback variable in the shared module to the value that came from the Entry widget. AND we don't actually destroy the child form. We just hide it:

```
def on_btnExit():

    print('ChildForm_support.on_btnExit')

    sys.stdout.flush()

    shared.passback = PassBack.get()

    MainForm_support.show_me()

    hide_me()
```

The next three functions should be pretty much familiar to you so we won't spend any time on them. Just be sure to add them:

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

def show_me():
    global root
    root.deiconify()

def hide_me():
    global root
    root.withdraw()
```

That's it for the code for the `ChildForm_support.py` module. Save your code, and then run the `MainForm.py` program. You can open and close the child form as many times as you want and everything should work correctly, including the number of times the child form has run and the data that's passed back from the child form to the main form.

Section 2 – Creating a Splash Form

Now that you know how to create programs that have two forms, let's add another one. The new form will be a splash screen that will be the first form that the user sees and will only stay on the screen for a few seconds.

What you will learn:

- How to create and use a splash form
- How to use the root.after() function as a timer
- How to embed graphics into a form

Here is what our new GUI form will look like after the design process is finished. At first glance, it might seem difficult, but in reality, it is a very simple form:

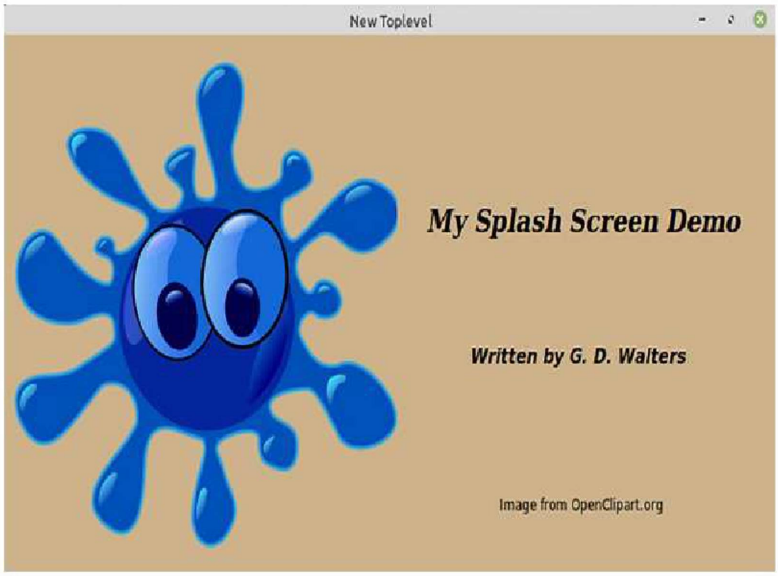


Figure 2.12: What our splash screen should look like.

Copy the entire folder from the DualForms folder and rename the copy "Splash". Next, create a new sub-folder and name it "graphics".

I am in no way, shape, or form an artist. For most of my graphics, I rely on the wonderful website OpenClipart.org. The website was down for a very long time due to issues but is now back up and ready for you. It is very easy to search for just about anything that you want to find. All of the artwork is available for any use that you need it for. Be SURE to give attribution to the artist and the website in your work, even if the artist is anonymous.

Designing the GUI

There are just 4 static Label widgets on our form this time. The first label will hold our graphic image. The other three simply hold the application title, the author's name, and a simple plug for OpenClipart.org. Once again, we'll start by adding a new `Toplevel` form. Set the Alias to **Splash**. Move it somewhere on the screen that is convenient (We'll use my `centre_screen` function to move it when the program runs). I made my form 798 pixels by 450 pixels. We also need to set the background color to "**navajowhite3**".

At this point, let's add our first label. This will hold our graphic. Here are the attributes you will need to set:

Attribute	Value
Alias	lblImage

Attribute	Value
background	navajowhite3
compound	none
justify	center
anchor	w

Now, we want to set the image attribute of the label to the filename of our graphic:

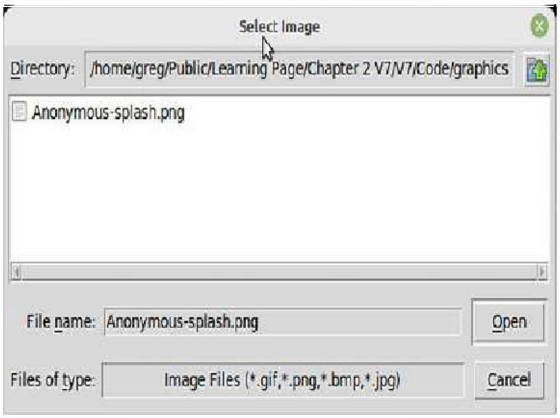


Figure 2.13: Label Attributes

Now, we want to set the image attribute of the label to the filename of our graphic:

If you look closely, you will see the image attribute is highlighted in a plumb color. Click on the button with three dots next to it. You will see a dialog that will allow you to select the image. You don't even need to type the filename! Please refer to the following figure:

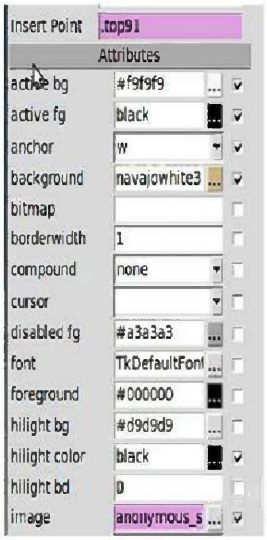


Figure 2.14: Select Image Dialog

Once you have selected the image in the dialog box and clicked the **Open** button, you should see the image in the label. Here is what your form should look like at this point:

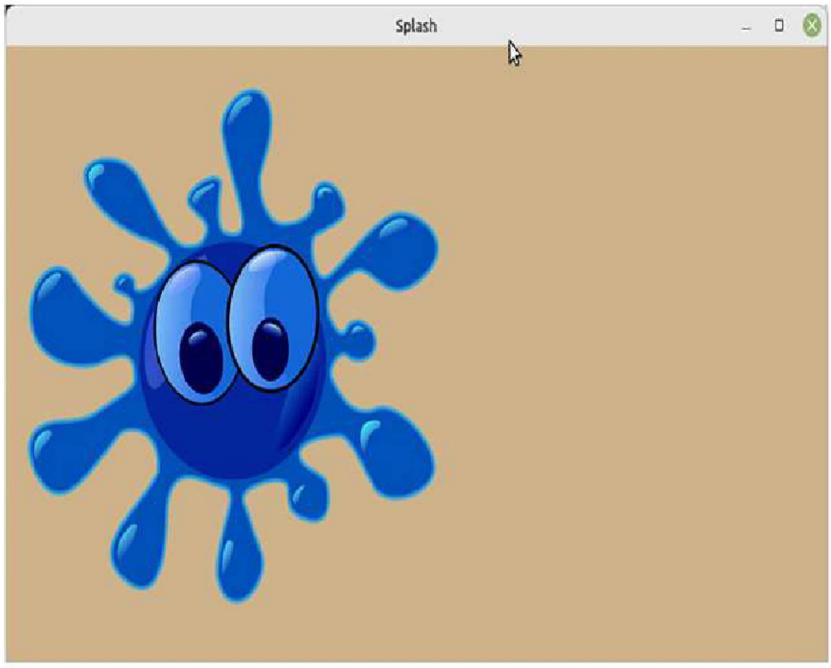


Figure 2.15: Our form so far.

Take this opportunity to save your project before moving on.

Your next step is to place the other three labels in your form somewhere on the right-hand side of your form. Here are the attributes for each with the X and Y positions:

Attribute	Value
Alias	Label1
X	420
Y	120
height	49
width	356
text	My Splash Screen Demo
background	navajowhite3
font	DejaVu Sans
	18 pt bold Italics

The second label will have these attributes:

Attribute	Value
Alias	Label2
X	430
Y	240
height	23
width	240
text	Written by G. D. Walters
background	navajowhite3
font	DejaVu Sans 12 pt bold Italics

Finally, here are the attributes for our third label. We won't set the font on this one:

Attribute	Value
Alias	Label3
X	439
Y	355
height	21
width	316
text	Image from OpenClipart.org
background	navajowhite3

That's all we need to do in the PAGE designer. Compare your form to the finished form as shown in (*Figure 2.11*). If everything is done, then we'll move on to modifying our code. Just so you know, when the splash form shows, the title bar won't be shown, so you really don't need to worry about setting the title attribute of the form. However, you should do it, just to keep in the habit of setting the form title.

The Code

Since we are using the code from the project at the beginning of the project, I will only show you the parts that need to be modified. We'll start with the main function. I'll make the parts you need to change in bold:

```
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 = DualForms.MainForm(_top1)
    # Creates a toplevel widget.
    global _top2, _w2
    _top2 = tk.Toplevel(root)
    _w2 = DualForms.ChildForm(_top2)
    # Creates a toplevel widget.
    global _top3, _w3
    _top3 = tk.Toplevel(root)
    _w3 = DualForms.Splash(_top3)
    startup()
    root.mainloop()
```

When you look at the main function, you will see that the code for the splash form comes AFTER the call to the **startup** function. This happens because PAGE looks for the line "**root.mainloop()**" and places the new code just before it. You will want to cut the call to the **startup** function and paste it just before the call to **mainloop** as shown previously:

```
def startup():
    shared.childrunning = False
    shared.childruncount = 0
    shared.passback = ''
    hide_Main()
    hide_Form2()
    _top3.overrideredirect(True)  # Hide the 'decorations'
```

```
    delay = 10000

    timerID = _top3.after(delay, on_tick)

    centre_screen(798, 450, _top3)
```

There are four new lines that need to be added to the end of the main function. I'll explain what each does.

The first line removes the title bar (properly called the **form decorations**) which keeps the form from being able to be moved or **Modal**. It will stay on top of any other forms until we remove the form.

The next time sets a variable named **delay** to 10000. This value is how long we want the splash screen to show. It is in **milliseconds**, which means we will show the splash screen for 10 seconds.

Next, we define a callback for a special Tkinter function properly called `root.after`. This creates a special timer that will call a function every N number of seconds. In this case, 10 seconds. This is how we will hide our splash screen and start the rest of the program. The `root.after` function can become a powerful tool in your GUI toolbox.

Finally, we call a special function that I wrote which will center a form within the user's screen. Since we don't know in advance what size screen our ultimate user will have, we'll use some special calls to get that number dynamically.

Here is the code for our `centre_screen` function. We pass in, as parameters, the width and height of the form and the actual object that is our form:

```
def centre_screen(wid, hei, which):

    ws = which.winfo_screenwidth()

    hs = which.winfo_screenheight()

    x = (ws / 2) - (wid / 2)

    y = (hs / 2) - (hei / 2)

    which.geometry('%dx%d+%d+%d' % (wid, hei, x, y))
```

The first thing we do is call the **winfo_screenwidth** and **winfo_screenheight** functions that Tkinter provides. These two functions (as their names suggest) will return the width and height of the user's screen. Then, we take each of these values and divide them by two to get the center of the screen. Next, we take the width and height (wid, hei) of the form and divide them by two, giving us the center of the form.

Finally, we subtract the center of the form values from the center of the screen values which gives us the x, y positions of the form. We then use the Tkinter **geometry** function to place the form. All of the Tkinter functions here are form specific, which is the variable that we pass in. It's the actual object that holds the form.

The `on_tick` function here is very simple. Basically, when the timer (10 seconds expressed as milliseconds) expires, a special **timer event** fires which calls our `on_tick` function, which contains code to do some work. In this case, it simply hides our splash form and shows the main form:

```
def on_tick():

    # ================================================

    # Once the 'timer' hits, hide the splash screen

    # ================================================

    hide_Splash()

    # ================================================

    #   Call the ShowMe routine in the main support file

    # ================================================

    show_Main()
```

All of the other code that we have already created stays the same. All we need to do is duplicate the show and hide functions and make them specific for our splash screen which is named `_top3`. I've called them `show_Splash` and `hide_Splash`:

```
def show_Splash():

    _top3.deiconify()


def hide_Splash():

    _top3.withdraw()
```

That's it. Save all your code and try running your program.

Now, I need to give you a small warning. If your computer has dual screens, the operating system (as mine does) might show the width as a combination of both screens and the height as the biggest screen. My primary screen is 2560 by 1440 and my second screen is 1680 x 1050. When I run the `centre_screen` function, the values it calculates are Width: 4240 - Height 1440. When the splash screen is shown, it ends

up almost fully on the right-hand side of the screen. There is a way to deal with this, which is simply to force the **ws** to (in my case) 2560 and the height (**hs**) to 1440. Most of your users will have a single-screen setup, so this is not much of a concern.

Conclusion

In this chapter, we learned how to create a project with multiple forms, how to create a splash screen, and how to use the `root.after` function.

In the next chapter, we'll take a look at all of the standard Tk widgets that PAGE provides, what you might want to use them for, and some of the special attributes that you might use to get the most out of them.

Multiple Choice Questions

1. Which attribute of a Label widget do you use to display an image in a PAGE form?
   a. Alias
   b. Compound
   c. Image
2. How do you make a static label into a dynamic label?
   a. Use the text variable attribute.
   b. You can't - you have to use a message widget.
   c. You can't – you have to create your own widget.
3. What Tkinter function do you use to create a timer?
   a. winfo.screenheight
   b. root.after
   c. geometry

Answers

1. c
2. a
3. b

Questions

1. How would you create a function that will show all forms of a multi-form project?
2. Why bother with a multi-form project when you could just create a single form and reuse the widgets that are already there?

Answers to the questions

1. The function would look something like this

```
def show_all():
    show_Main()
    show_Form2()
```

2. There are many times that a single form just cannot hold all the widgets that would be needed to eliminate the use of multiple forms. In addition to that, let's say that you want to show special configuration variables. Attempting to put them on the main form with everything else, would most likely confuse the end user. There is a solution, however. You could use a ttk TNotebook and place all the widgets for each form into separate pages (or tabs). That would eliminate the need for multiple forms.

Key terms

- root.after
- root.winfo_screenwidth
- multi-form project

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3
Standard Tk Widgets

Introduction

In the previous chapters, you were able to learn the basics of using PAGE to create GUI programs. In this chapter, we will look at each of the standard widgets in the PAGE toolox that can go into your GUI projects. We will take a look at the most important attributes that control each of the widgets.

What you will learn

In this chapter, you will learn the following topics:

- Common Attributes
- Geometry
- Toplevel Widget
- Button Widget

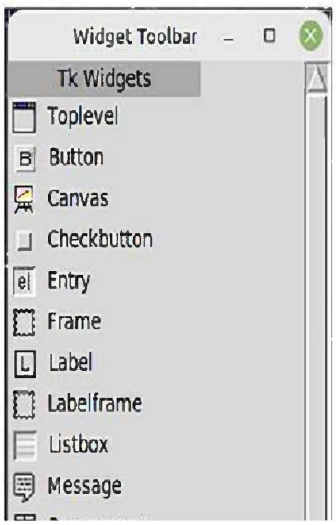
- Button Widget
- Canvas Widget
- Checkbutton Widget
- Entry Widget
- Frame Widget

- Label Widget
- Labelframe Widget
- Listbox Widget
- Message Widget
- Radiobutton Widget
- Scale Widgets
- Spinbox Widget
- Text Widget
- Popupmenu

Tk Widgets

PAGE supports 16 standard Tk widgets plus a form menu. We have already used a few of these widgets, but we will explore all these widgets in depth and look at some of the most useful attributes of each so you can get the most out of each one.

After we go through the common attributes, I will explain each of the widgets in the same order that they are on the toolbar:



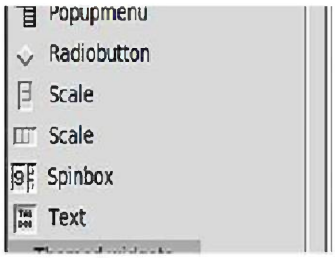


Figure 3.1: Tk Widgets in the Toolbar

> *Note: We will be using the term 'Focus' many times in this chapter (and upcoming chapters).*

In GUI programming, a widget gets focus by either clicking on it or the user can use the tab/shift tab key on the keyboard to move through the widgets. PAGE creates what I call the Tab Focus Tree (officially called focus traversal) based on the order in which you add widgets. If a widget has focus, certain attributes will come into play that is not evident under normal circumstances. If you move a widget on the designer form after other widgets have been placed, the original location in the Tab Focus Tree will still be in effect. There is a way to change the Tab Focus Tree that we will explore in an upcoming chapter.

To find out more information about the Tk widgets, you can visit **https://www.tcl.tk/man/tcl/TkCmd/contents.html**.

Common attributes

There are some attributes that almost every Tk widget has in common.

Background Color: This is the color of the BODY of the widget.

Foreground Color: This is the color of the TEXT of the widget. This is only available for widgets that support text.

Border width: The border is the outline around the widget. The border width is measured in pixels.

Cursor: This is the shape of the mouse pointer. You can set this by the entire Toplevel (form) or on an individual widget basis. There are some cursors that are available only for certain operating systems while others are common among all operating systems. This is important to remember if you are planning on having your program run on multiple operating Systems. Refer to **https://www.tcl.tk/man/tcl8.4/TkCmd/cursors.html** and for a visual reference of the cursors, refer to **https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/cursors.html**.

Highlight background: This is the color of the highlight border when the widget

Highlight background: This is the color of the highlight border when the widget does not have focus. This is the left and top sides of the border.

Highlight color: This is the color of the highlight border when the widget has focus. This, again, is the left and top sides of the border.

Highlight border width: This is the border width of the highlight border.

X pad: This is the amount of space added to the image or text's left and right. Usually, this is the number of pixels and you will simply use an integer value. However, you

can use something like "3m" for millimeters. The other options are c for centimeters, i for inches, and p for printer's points. More times than not, you will simply use an integer for the number of pixels.

Y pad: Similar to the X pad, this is the amount of space added to the top and bottom of either the image or text.

Relief: This allows you to change the 3D look of the border around the widget. The following image shows the way the different relief setting effects a Label widget, but for the most part, the same effect applies to almost any Tk widget:

Figure 3.2: Relief options for the Label widget

Take focus: You often want a widget to be in the focus tab tree. The Tk Frame, Label,

Labelframe, and Menu widgets normally are not included in the focus tab tree. You can take any widget out of the focus tab tree by setting the `takefocus` attribute to 0. If you want to include a widget that is not normally in the focus tab tree, set the

`takefocus` attribute to 1.

Text Variable (`text var`): While not every widget supports this, almost every widget that supports text has this attribute. This gives you a very easy way to set and get the text from a widget. To set the text for a widget, use the .set() method. To retrieve the text from the widget (like an Entry widget), use the .get() method. The `text var` attribute allows you to name the variable but each one will be a special Tk Variable. These variables can be a type `StringVar`, `IntVar`, and `DoubleVar`. The variables you create for this attribute can be found in the project.py file near the top of the class for that Toplevel.

Geometry

Geometry refers to the position and size of the widget.

X position: This is where the widget is placed within its container on the left/right axis.

Y position: This is where the widget is placed within its container on an up/down axis.

Width: This is the width of the widget in pixels.

Height: This is the height of the widget in pixels.

The next four attributes come into play only when PAGE is in the relative mode.

> *Note: Do not change these values by hand. These values are calculated by page. If you change them by hand, things can go bad.*

Relative x: This is the relative percentage of the widget's left/right position to the container size.

Relative y: This is the relative percentage of the widget's up/down position to the container size.

Relative width: This is the relative percentage of the widget's width to the container size.

Relative height: This is the relative percentage of the widget's height to the container size.

Now, we will look at the various Tk widgets one by one. Many attributes of the widgets have the same attributes as other widgets and will for the most part, do the same thing.

Toplevel

The Toplevel widget is a container that will hold all the other widgets that make up your GUI. In this book, I will refer to the Toplevel widget as the "form". Think of the Toplevel as a painter's canvas that eventually will hold their work of art.

There are some important attributes for the Toplevel widget.

Alias: The alias of the Toplevel will become the name of the class that makes up the GUI class. By default, PAGE uses an alias of Toplevel1 for the first Toplevel in your project, Toplevel2 for the second, and so on. A good alias name for the main Toplevel form would be "Main". If your project has more than one Toplevel, you could set the

alias for the second to "Second" or "Configuration" if that is what that form will be used for.

Background: This is the color that you want your form to have.

Menu: By clicking on this Attribute, you will be presented with the PAGE Menu Editor. This will create a Text menu just under the title bar of your form. We will be looking at Menus in *Chapter 7, Menus and Pop-Up Menus*.

Title: This is the text that appears along the top bar of your form. It is important to create a title that relates to the purpose of the form. For the main form of your project, this is usually something like the name of your program. The title bar is also known as the decorator bar. The title is often static (does not change during the running of the program), but can be modified within your code.

These three attributes are usually the only ones you will set. To be perfectly honest, over the 10+ years that I have been using PAGE, I have only used any of the other attributes may be less than 10 times.

Button

The Button widget is used to allow the user to start an action. When the user clicks on a Button widget, this causes an event to be raised that allows a certain process to start or end. The Button widget can display text, an image, or both. PAGE supports about 30 different attributes for the Button widget. We will not discuss all the attributes, but here is a list of the ones you would most likely use:

Figure 3.3: Button in Toolbox

Active Background (`active bg`): This is the color of the button when the user moves the mouse over the widget.

Active Foreground (`active fg`): This is the color of the text of the button when the user moves the mouse over the widget.

Anchor: If you are using text on your Button widget, this specifies where the text will show up on the face of the button. The options available are n, ne, e, se, s, sw, w, nw, and center. You can think of these options as points of a compass.

Background: This is the color of the face of the Button.

Bitmap: You can use a special type of image. You will not use this attribute very often, if ever, due to the fact the options are fairly limited and to be brutally honest, look very "old school".

Border Width: The border of any widget is the decoration around the outside of the widget. Usually, you will not change this for the Button widget and defaults to 2 pixels.

Command: This is the name of the function that will be called when the user clicks on the Button. This is called a callback function. We have already used the button widget in the previous two chapters. Normally, you will just provide the name of the function without any ending parentheses. When you set this attribute, PAGE will create a function skeleton for you in the Support module.

Compound: When you use an image on a Button, this attribute sets the location of both the text and the image. Sometimes, you do not need to add text if the graphic image is obvious to the intent of what the Button will do when the user clicks on it. Sometimes, however, you might want to include text to provide extra help to your user. The options are none, bottom, center, left, right, and top. If you want just text or just an image, use "none".

Disabled Foreground: This is the color of the text when the state of the Button is set to "disabled".

Font: This attribute provides a dialog to allow you to easily set the font, font size, and the style of the font (Bold, Italic, and so on.)

Foreground: This is the color of the text. You can enter the hex value of the color, a color name or, if you click on the button just to the right of the attribute, will provide a dialog box that allows you to pick the color.

Image: Clicking on this attribute will bring up a dialog box asking for the name and location of the image you wish to use. Tkinter allows you to use the following image file types *.gif, *.png, *.bmp. In PAGE 7.x, you can also use *.jpg files. There are pros and cons to the various types of images that you can use. Under most circumstances, you will probably use the *.png file type, since this type of file supports a transparent background of the image.

Justify: When you have multiline text, this sets where the text is aligned. The options are left, right, and center.

X pad: This is the amount of space added to the image or text's left and right. Usually, this is the number of pixels and you will simply use an integer value. However, you can use something like "3m" for millimeters. The other options are c for Centimeters,

i for inches, and p for printer's points. More times than not, you will simply use an integer for the number of pixels.

Y pad: Similar to the X pad, this is the amount of space added to the top and bottom of either the image or text.

Relief: This allows you to change the 3D look of the border around the widget.

State: This allows you to define if the widget is disabled (cannot be clicked on), normal or active. Usually, you will leave this attribute set to the default, which is normal.

Text: The text that appears on the widget.

Text Variable (`text var`): This attribute allows you to set a variable that allows you to change (or obtain) the text easily within your code.

Tooltip Text: A Tooltip is a small box that pops up above the widget when the mouse hovers over the widget for a few seconds. The text you enter here allows you to provide some extra help about the use of the particular widget.

Canvas

The Canvas widget is a very powerful widget but one that you will not use very often under normal circumstances. If you need to generate your own graphs and plots, this allows you to do it, but it is very complicated. We will look at using the Canvas widget in *Chapter 11, Using the Canvas Widget*. The Canvas widget can also act as a container for other widgets. There are a number of attributes for the Canvas

widget, but we will only look at a few of them, since this widget is really only useful as a container, unless you write a large amount of code to utilize it:

Background: This is the color of the face of the widget.

Figure 3.4: Canvas widget in Toolbox

Border width: This is the size of the outline around the widget in pixels.

Relief: This allows you to change the 3D look of the border around the widget.

Checkbutton

The Checkbutton is a very useful widget that allows you to provide a simple Yes/No or On/Off style indicator. Like the Button widget, you can provide a callback

function. We have already used the Checkbutton in an earlier project. As said, it's a very useful widget and there are many different ways you can use it:

Active Background: This is the color of the widget when the user moves the mouse over the widget.

Figure 3.5: Checkbutton in Toolbox

Active Foreground: This is the color of the Text of the widget when the user moves the mouse over the widget.

Anchor: If you are using text on your widget, this specifies where the text will show up on the face of the widget. The options available are n, ne, e, se, s, sw, w, nw, and center. You can think of these options as points of a compass.

Border width: The border of any widget is the decoration around the outside of the widget. Usually, you will not change this for the Checkbutton widget and defaults to 2 pixels.

Command: This is the name of the function that will be called when the user clicks on the widget. This is called a callback function. Normally, you will just provide the name of the function without any ending parentheses. When you set this attribute, PAGE will create a function skeleton for you in the Support module.

Compound: This attribute allows you the change the position of the indicator or image in relation to the text.

Indicator: This attribute controls whether the small check indicator is visible or not. Normally you would keep this to the default of Yes. However, in special instances, you can turn it off and provide your own.

Image: This attributes allow you to provide your own version of the small check indicator in the Off state.

Relief: See the definition near the beginning of the chapter.

Select Image: Like the Image attribute, this allows you to provide your own image for the On state.

State: This allows you to define if the widget is disabled (cannot be clicked on), normal or active. Usually, you will leave this attribute set to the default, which is normal.

Text - The text that appears on the widget.

Text Variable (text var): This attribute allows you to set a variable that allows you to change (or obtain) the text easily within your code.

Variable: This attribute provides you the ability to set or check the value of the widget. PAGE provides a default value for you, but you can change it here. This value should be unique and the name should not contain any spaces. If you copy a Checkbutton and paste it within your project, all the attributes you have set will be copied as well, including the name of the original variable. Be sure to set it with a unique value.

Entry

The Entry widget is designed to allow your user to enter a single line of text or numbers. It does not support multi-line text. You can use the Text widget or PAGE's ScrolledText widget for multi-line text. There are a number of attributes that are not used very often, so we will not discuss them here. The following image is an illustration of an Entry Widget:

Figure 3.6: Entry Widget in Toolbox

Background: For the Entry widget, this is the color of the text entry area. The default is white.

Disabled Background (`disabled bg`): This is the background color when the Entry widget is in the disabled state.

Disabled Foreground (`disabled fg`): This is the foreground (Text) color when the Entry widget is in the disabled state.

Export Selection: This is Yes/No attribute. If your user selects any text within the Entry widget, it is, by default, automatically copied to the system clipboard. While this can possibly be a good thing, there are a number of widgets that support this attribute and can cause big problems for your program. I would suggest that you set this to No manually whenever you use an Entry widget.

Readonly Background (readonly bg): This is the color of the Text entry area (not the text) when the widget is in the readonly state.

Show: The widget normally displays the characters typed. You can use the Show attribute to mask the characters being typed to another character. For example, a password entry field should not display the characters of the data entered such a field. You can set this attribute to * to hide the characters entered.

State: This attribute sets the state (or ability to focus and make changes) of the widget. The options here are normal, disabled, and readonly.

Readonly Background (readonly bg): This is the background (Text area) color when the widget is in the Readonly state.

Text Variable (text var): This attribute allows you to specify a variable name that can be used to set or get the information that is in the Text Entry area. This is only the name of the variable.

Validate: This attribute can be used to allow you to check the validity of the data that was entered into the widget. This is just one part of the attributes that need to be set. `Validate cmd` and `inv. cmd` are also used.

Frame

The Frame widget is simply a container for other widgets. You can use the frame to group widgets together:

Background: This is the color of the face of the frame.

Figure 3.7: Frame widget in toolbox

Border width: This is the number of pixels that make up the border of the widget

Border width: This is the number of pixels that make up the border of the widget.

Highlight Color (`hilight color`): See Common Attributes

Highlight Background (`hilight bg`): See Common Attributes

Highlight Border width (`hilight bd`): See Common Attributes

Relief: See Common Attributes

Take Focus: Normally, a Frame and Labelframe will not be in the Tab Focus Tree. The default for this attribute is 0. If you want the Labelframe to be included in the Tab Focus Tree, set this attribute to 1.

X Pad: This allows a given number of pixels padding around the inside border of the Label widget horizontally.

Y Pad: This allows a given number of pixels padding around the inside border of the Label widget vertically.

Label

The Label widget is normally considered a way to display text that will not change, like the definition of a widget next to it. This is usually called a Static label. However, a Label widget can also be dynamic, in which you can change the text via your code. A Label can also display a graphic image alone or with text like a button. The attributes for the Label widget are almost the same as for the Button widget, except there is no command attribute to set a callback. Also, the Label widget normally cannot get focus:

Figure 3.8: Label in Toolbox

Active Background: This is the color of the widget when the user moves the mouse over the widget.

Active Foreground: This is the color of the Text of the widget when the user moves the mouse over the widget.

Anchor: If you are using text on your widget, this specifies where the text will show up on the face of the widget. The options available are n, ne, e, se, s, sw, w, nw, and

center. You can think of these options as points of a compass.

Bitmap: You can use a special type of image. You will not use this attribute very often, if ever, due to the fact the options are fairly limited and to be brutally honest, look very "old school".

Compound: This attribute allows you the change the position of the indicator or image in relation to the text.

Font: This attribute provides a dialog to allow you to easily set the font, font size, and the style of the font (Bold, Italic, and so on.)

Image: As I stated above, the Label widget can be used to display text, a graphic image, or both. Clicking on this attribute will bring up a dialog box asking for the name and location of the image you wish to use. Tkinter allows you to use the following image file types *.gif, *.png, *.bmp. In PAGE 7.x, you can also use *.jpg files. There are pros and cons to the various types of images that you can use. Under most circumstances, you will probably use the *.png file type, since this type of file supports a transparent background of the image.

Justify: This attribute only comes into play when you have multiple lines of text in a Label widget. The options are left, right, and center.

Relief: See Common Attributes

State: This allows you to define if the widget is disabled (cannot be clicked on), normal, or active. Usually, you will leave this attribute set to the default, which is normal.

Take focus: Normally, a Label is not included in the Tab Focus tree. To allow a Label to take focus, set this attribute to 1.

Text Variable (text var): This attributes allows you to set a variable that allows you to change (or obtain) the text easily within your code.

X pad: See Common Attributes.

Y pad: See Common Attributes.

Wrap length: This attribute sets the number of characters that can appear on a single line. The default is 0 which means lines will only be broken at newline characters. You can set this attribute to 1 to emulate a vertical label. The only problem with this is that space characters will not be displayed in the emulated vertical mode.

Labelframe

The Labelframe is just like a normal frame with the exception that it has a build in Label. You can set the text by using the Text attribute. There is no text var attribute, so while you can change the text in your code when you need to, you would need to use the `widgetname.config(text="xxx")`

Background: This is the color of the face of the frame:

Figure 3.9: Labelframe in Toolbox

Border width: This is the number of pixels that make up the border of the widget.

Foreground: This is the Text color for the label of the widget

Anchor: The position of the label around the border of the frame. The options available are n, ne, e, se, s, sw, w, nw, and center.

X Pad: This allows a given number of pixels padding around the inside border of the Label widget horizontally.

Y Pad: This allows a given number of pixels padding around the inside border of the Label widget vertically.

Take Focus: Normally, a Frame and Labelframe will not be in the Tab Focus Tree. The default for this attribute is 0. If you want the Labelframe to be included in the Tab Focus Tree, set this attribute to 1.

Text: This is the text of the Label portion of the widget.

Listbox

The Listbox widget is an easy way to show a series of lines of text that the user can select from. All the lines of text use the same font. The user can use the up and down arrows on the keyboard to "scroll" through the list of items if there are more items than can be displayed in the space allowed. By default, only one item is selected at a time, but multiple items may also be selected at a time (see selectmode). The following figure represents Listbox.

Active style: This attribute allows you to specify the way the active line looks visually. There are three options here. Underline displays the active line underlined. Dotbox encloses the active line with a dotted box on all four sides. None does nothing special for the active line.

Figure 3.10 - Listbox widget in Toolbox

Export Selection: Like the Entry widget, this automatically places the selected information on the clipboard. This can cause unexpected issues, so normally you would set this to No.

List var: This is a special variable name that is connected to the list of items. However, the variable is connected to the entire list. Similar to the text var attribute, you can use the `.set()` and `.get()` methods; however, remember this deals with the entire list at one time.

Selectmode: This attribute allows the manner of selection. While there are four modes (BROWSE, SINGLE, MULTIPLE, and EXTENDED), usually you would use BROWSE or SINGLE which only return a single item. MULTIPLE and EXTENDED allow for multiple selections to be returned. The .get() method always returns a tuple.

Message

The Message widget is like the Label widget but is used when you need to display text with multiple lines as shown in *Figure 3.11*.

For the most part, the attributes for the Message widget are the same for the Label widget.

Figure 3.11: Message Widget in toolbox

Radiobutton

The Radiobutton is similar to the Checkbutton except it is designed to be used in a one-of-many option sets. Think about an old-style car radio which allowed you to preset your favorite radio stations to a few push buttons. When you push one, the others are no longer selected. When you use Radiobuttons, you should usually place them inside a frame to visually group them together.

Figure 3.12: Radiobutton in the toolbar

Variable: This is the name of the variable that will hold the value of the currently selected Radiobutton. PAGE provides a default of selectedButton. Change this to something that suits your needs. Each Radiobutton in a group should share this variable.

Value: This is what will be passed to the variable name. This can be a simple integer (1, 2, 3, and so on.) or a string ('Apples', 'Bananas', 'Oranges', and so on).

Command: Like the command attribute for the Button widget, this holds the name of the callback function that will be triggered when a Radiobutton is selected. Usually, your entire group would have the same callback name.

Image: Like the Checkbutton, you can create custom indicators for selected and unselected states. The image attribute is for the unselected image.

Indicator: Again, like the Checkbutton this attribute should be set to Yes to use the normal dot in a circle. If you wish to use custom graphics, set this to No

SelectImage: This holds the filename of the selected image.

Scales

There are two versions of the Scale widget. One horizontal and one vertical. These allow you to set a value, either integer, or float within a preset range.

Each Scale has a slider "button" that can be dragged back and forth to change the value. The left arrow and up arrow keys will move the slider up for the vertical scales or left for horizontal scales and the right arrow and down arrow keys will move the slider down or right. *Figure 3.13* is an illustration of the Vertical Scale widget:

***Figure 3.13**: Vertical Scale widget in the toolbox*

Command: This is the name of the callback function to be triggered whenever the scale value changes.

Digits: When you retrieve the current value of the scale, this sets the number of digits to use when the value is converted to a string.

Resolution: Normally, the scale will only change in whole integer units (-1, 0, 1). If you need to change the slider in smaller increments, you can set this to a smaller

value. If, for example, you need to deal with the values from 1.0 to 1.0 and allow 1/2 value steps, you would set the Resolution attribute to 0.5, which would then allow for the outputs 1.0, -0.5, 0.0, 0.5, 1.0.

Show Value: This provides for a visual indicator to be displayed next to the scale Widget of the current value.

Tick Interval: If you want tick indicators to be shown along the Scale, you can set this. The default is 0, which hides the tick marks.

From: This is the lowest value the widget will accept. To set this value in code, you would use "from_" since "from" is a Python keyword.

To: This is the highest value of the range.

Variable: This is similar to the Text Var of a Button or Label widget. You can use the .set() method to set the current value and use the .get() method to retrieve the current value.

Spinbox

The Spinbox widget provides a stepped selection of a range of numbers or strings. There is an up and down arrow to the side that allows the user to click through each value.

Figure 3.14: Spinbox widget in toolbox

Command: This specifies the callback function which is triggered whenever the value changes.

Export Selection: This is like the Listbox and the Entry widgets; this places the current value on the clipboard.

From: This is the same as for the Scale. The lowest value of the range if using values to set this value in code, use "`from_`" not "`from`" since "`from`" is a Python keyword.

To: This is the highest value for the range (if using values).

Values: If using a set of strings to choose from

Wrap: By default, this is set to No, which won't allow the value to change once the either end of the range is met.

Text

This widget allows freeform display and entry of multiple lines of text. While this is a very useful widget, there are a great number of attributes and methods that make this a fairly complicated widget to work with. You can mix text with different fonts and colors and backgrounds, you can also show images with the text:

Figure 3.15: Text widget in toolbox

Many of the attributes available from the Attributes Editor, you will not use under normal situations, they will be done in your code.

Background: This is the color of the Text edit/display area. This defaults to white.

Export Selection: This is like the Listbox and the Entry widgets; this places the current value on the clipboard. While this can possibly be a good thing, there are a number of widgets that support this attribute and can cause big problems for your program. I would suggest that you set this to No manually whenever you use a Text widget.

Font - This sets the DEFAULT font for the widget. As I said earlier, you can use multiple fonts within the widget.

Foreground: This is the DEFAULT text color. You can have multiple text colors within the widget.

Select Background (select bg): This is the background color that is used when selecting text.

Select Border Width (select bd): This is the width of the border to use around the selected text.

Select Foreground (select fg): This is the foreground (Text color) to use when selecting text.

Wrap: This attribute will control the display of lines that are too long to display on a single line within the widget. The options are:

Char: Any line that gets longer than can be displayed on a single line will break and wrap at any character.

Word: This will cause the line to break at the last work that will fit on the line.

None: No wrapping will occur.

I suggest you set the wrap to Word by default.

> *Note: PAGE offers a Scrolledtext widget under the Enhanced Widgets section of the toolbar. The same options are available for the Scrolledtext widget as the Text widget, but offers Scrollbars. The programming is the same for the Scrolledtext widget as the Text widget.*

To learn more about programming the Text widget, you can find very good information at **https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/text.html**.

Popupmenu

PAGE allows you to create Popup Menus, also called Context menus, which are normally accessed by clicking on a widget with mouse button 3 (right mouse button for most people). When you click on the Popupmenu button in the toolbar, an empty popup menu is added to the Toplevel form which can be seen in the Widget Tree. You will *NOT* see any indication of the Popup menu on the Designer form itself. We will be exploring menus and popup menus in *Chapter 7, Menus and Pop-Up Menus*.

Figure 3.16: Popupmenu in Toolbox

All Tk widgets in action

For a number of releases of PAGE, we have been creating a Widget Demo that helps us try to break the early test versions of PAGE before it gets out to the world. The

way we look at it, if we can put every widget in the toolbox into a program, and nothing breaks, it is probably 99% bug free.

We start with a ttk Notebook widget (which we will talk about it in a later chapter), which has a tab or page for each type of widgets. Tk widgets, ttk widgets, and enhanced widgets. Then, we create a special final page on the Notebook that we do some fun things that most programmers do not know about. Kind of a "special bonus page".

As represented in *Figure 3.17*, see all the TK widgets in action:

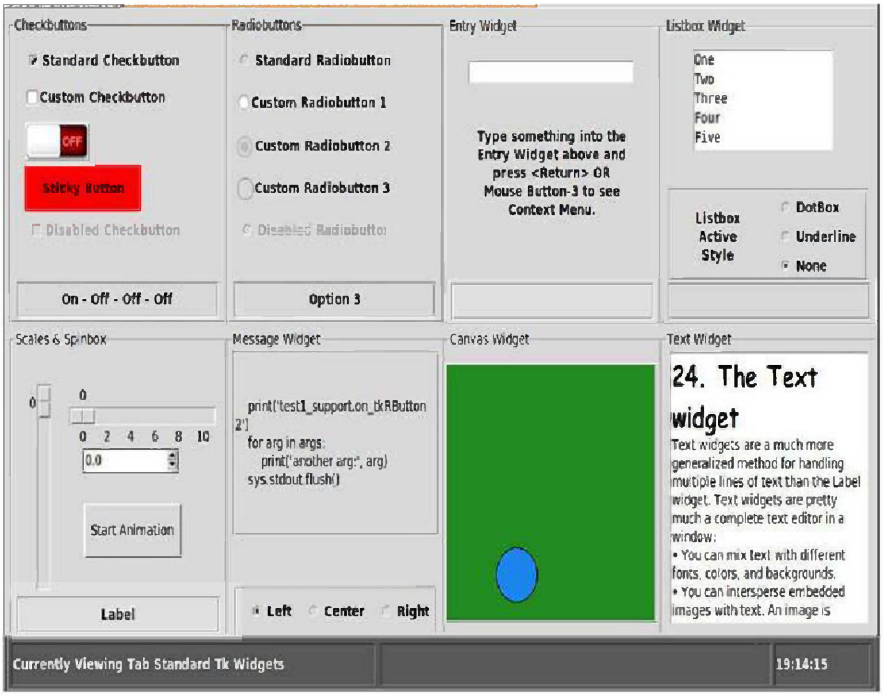


Figure: 3.17: All Tk widgets in action

For example, most programmers do not know that if you do not like the default check mark for the Checkbutton or the default little dot for the Radiobutton, you do not have to use a ttk Checkbutton or ttk Radiobutton to get something different. You can provide your own graphics for the Standard Tk Checkbutton or Radiobutton.

This gives a quick visual example of all the Tk widgets all at one time. I have got a notebook page for each of the different types of widgets that we will use as we move along through the book.

Conclusion

In this chapter, we learned about the various Tk widgets that PAGE supports, their important attributes, and some general reasons to use each.

In *Chapter 4, Pinger Program*, we will look at using some of these widgets in an unusual way.

Multiple Choice Questions

1. The Message widget is the exact same thing as the Label widget.

a. True

b. False

2. Frames and Labelframes are used:

   a. To group like widgets together

   b. To draw attention to a group of widgets

   c. To group together a number of widgets that should logically go together as a GUI "thought"

3. Labels are just for static text.

   a. True

   b. False

Answers

1. B – False. While the Message widget and the Label widget are very similar, they are not the same. You would use the Message widget when you need to have multiple lines of text displayed to the user.
2. C – You can group widgets together in a logical group. They all do not have to be the same type of widget.
3. B – FALSE. Labels can be both static and dynamic.

Questions

1. What is the major difference between the Radiobutton and the Checkbutton widgets?
2. Why would you use a Listbox Widget?
3. Which widgets are best for displaying graphic images?

Answers to the questions

1. Radiobuttons can be considered a one-of-many display options. Checkbuttons can be considered a Yes/No or many-of-many display option. While you can have many groups of Radiobuttons, only one Radiobutton in a group can be selected at any time.
2. A Listbox provides the user many options that appear in a list vertically. Many of the options can be viewed at once. Depending on the setting of the selectmode attribute, the user can select just a single item out of the list or multiple items. Many users are familiar with the concept of a dropdown or combo box where only one item is displayed until the widget is clicked and then a number of

items will "drop down" while the widget has focus. A Listbox always shows multiple items.

3. Many widgets support images or graphics, but under normal circumstances, the Label widget is usually the best choice to display an image. Buttons are great for showing graphics that are used instead of text, like an Exit button.

Key terms

- Focus
- Static
- Dynamic
- Absolute Mode
- Relative Mode

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

The Pinger Program

Introduction

In the first three chapters, we learned a great deal about PAGE and the standard Tk widgets available. In this chapter, we will create a program that will help verify your Internet connection by attempting to contact a website using a process called Ping.

What you will learn

In this chapter, we will learn the following topics:

- The History of Pinger
- Designing the GUI
  - Adding the Main Menu
  - Filling out the rest of the Main GUI
  - Adding the About Form
- The Code
- Creating a program that easily supports multiple languages
- Using Python to control an operating system utility
- Creating Log files

- Creating strings from a text file
- Substituting data in Strings using templates
- Controlling the colors of various widgets on the fly
- Using Python to control an operating system utility
- Creating Log files. Substituting

The History of Pinger

In 2017, my son and I moved into an apartment in a small college town. Internet access was important to both of us and we got the best connection speed that we could afford. Normally, things worked well and we could connect to any website we needed and could do the work we needed to do. Occasionally, however, there were issues. It seemed like the Internet Provider just went away for moments to hours at a time for no apparent reason.

We contacted the provider, but they would say that everything was working correctly from their perspective.

I decided to create a program that would provide a way to provide proof that we were having issues in the form of logs. I wanted to write the program in Python and wanted a GUI front end for it, so I turned to PAGE. Basically, I wanted to have a way to control the Ping process from the GUI and get back a log showing the number of successes and failures as well as the time that it took for the return response. I came up with this:

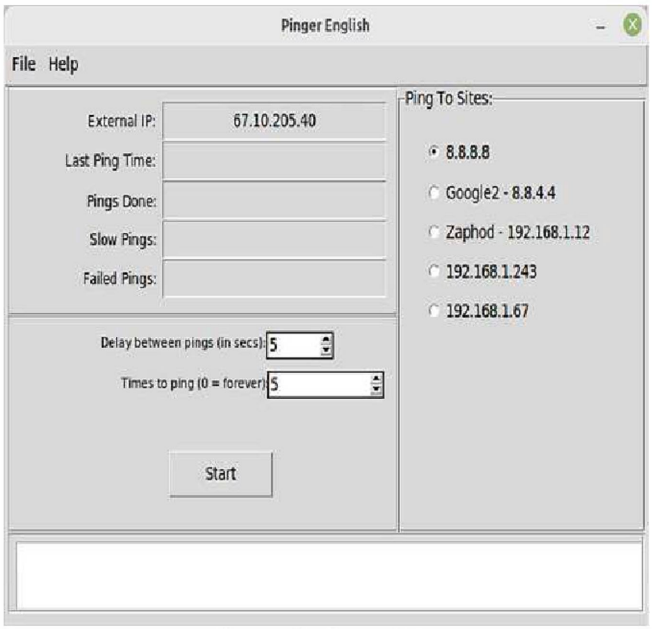


Figure 4.1: Pinger GUI

I admit that my first versions really didn't look much like this. There was no menu at the top and it was only written for me (I only speak two languages, English and BAD English), so there wasn't any support option for other languages.

Anyway, I got the program written and tested and the next time we started having issues, I used it to get some interesting data. We contacted our provider and gave them the data that my program provided. They came out and found that there was an intermittent connection coming into the apartment. They fixed that and that was the end of the issues.

At that point, I had a neat little program that worked fine but I did not need it any more. I decided to use it as a teaching tool for others. I have a friend in Norway and passed it along to him. He does speak English but natively speaks Norwegian. I thought it would be neat to provide the ability to keep the same layout and information but switch languages for the GUI to suit the user. Many years before, I had worked for a company that had a program that needed to support multiple languages and I decided to modify the way they handled that issue to make it easy for others.

There are a lot of imports, but almost all of them are standard Python library packages except the request module. You can add that to your Python installation by typing the following line into a terminal (or command prompt if you are in Windows):

```
pip install -upgrade requests
```

Designing the GUI

When we are all done, we will have two forms: the main form and a form for the About box. On the main form, there will be a simple form level menu, multiple labels, a frame with Radiobuttons, and a couple of Spinboxes that will allow us to set some parameters, a Start button, and a text area to display important information. The About form will simply have multiple areas to display information and a dismiss button.

Start PAGE and set the Toplevel form to 652 pixels wide and 605 pixels high. Set the alias of the form to "`pinger`" and the title to "`Pinger`".

Adding the main menu

Now, we'll create the main menu. On the Attribute Editor, click on the menu creation entry, as shown in *Figure 4.2*:

Figure 4.2: Attribute Editor Menu Editor Launcher for the Main Form

This will bring up the main menu editor. It should look something like this:

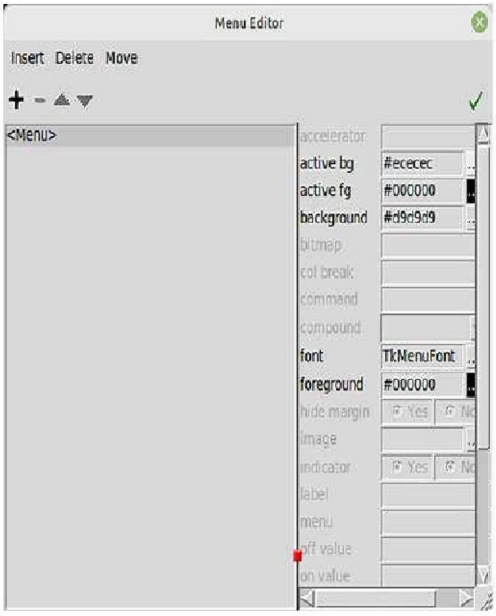


Figure 4.3: Menu Editor

To create a menu, we need to add items to it. The **Insert** and **+** icon will cause a drop-down menu that looks something like this:

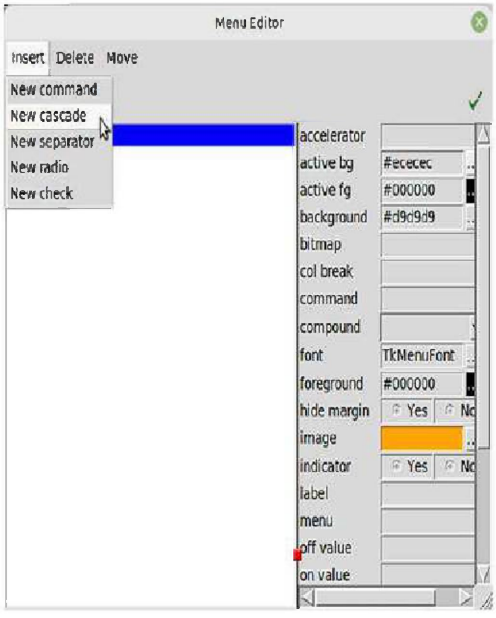


Figure 4.4: Menu Editor Insert a New Cascade Item

Before you can properly add items, you need to know the difference between Cascade menu items and Command items. If you take a look at *Figure 4.3*, you will see three main menu items (`Insert`, `Delete`, and `Move`). These are Cascade items. This means that they are like headers, with other items below them. In the case of the first menu item `Insert`, five commands can be performed below it (`New Command`, `New Cascade`, `New Separator`, `New Radio`, and `New Check`). For our project, we will keep the menu very simple:

```
File  (Cascade Item)

    |- Exit   (Command Item)

Help   (Cascade Item)

    | - About     (Command Item
```

So, to add a cascade item, start by clicking on the `<Menu>` item and then click on the + icon, and select New Cascade. The menu editor will now look like this:

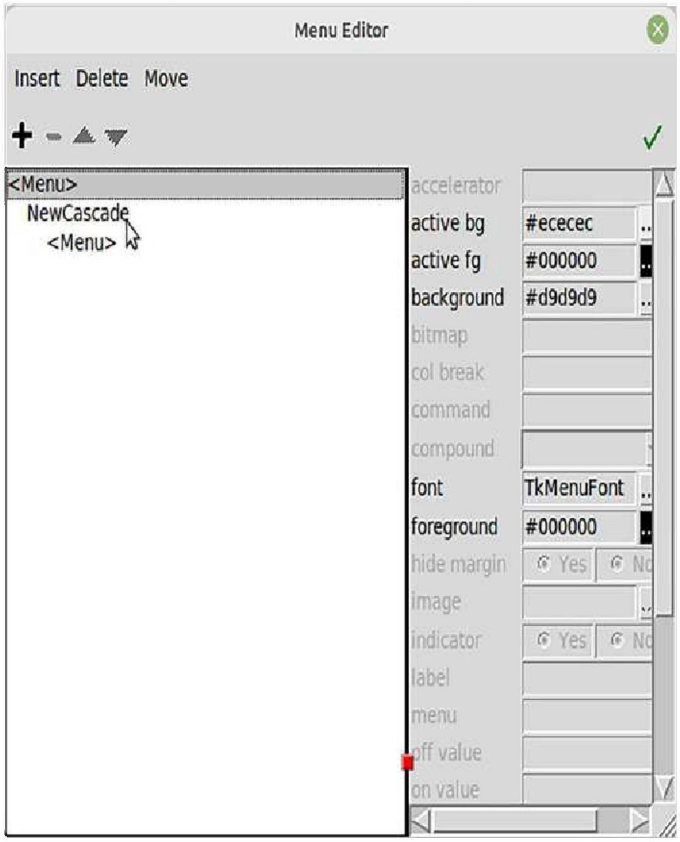


Figure 4.5: Adding a new Menu item

Now, click on **New Cascade** and click on the **+** icon again. This time select **New Command**:

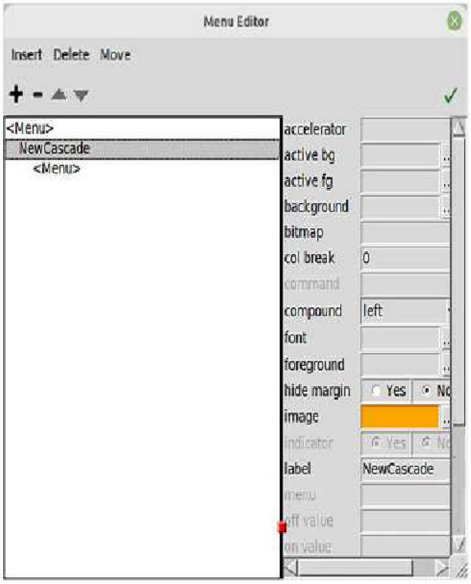


Figure 4.6: Adding a new command

Notice that the sidebar of the editor now can modify many attributes of the menu item like **label**, **background**, and more. This time, we will only change the label. Change it from **New Cascade** to **File**. Just like the widget Attribute Editor, the options will change depending on what type of menu item you select.

Now, with the **NewCascade** item selected, click on the **+** icon again and this time select the **New command**. Again, the side panel will change showing the items that we can change, as shown in *Figure 4.7*:

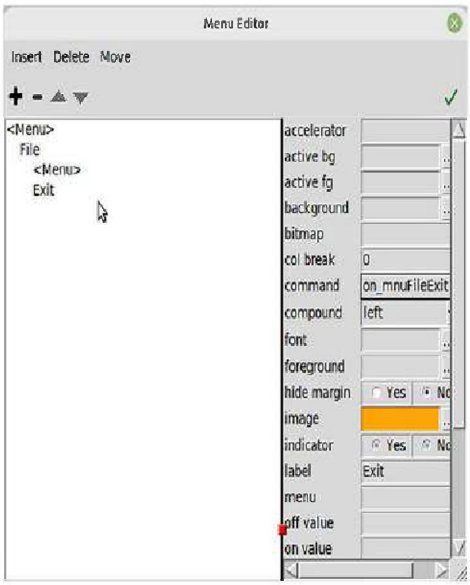


Figure 4.7: First Menu Item Finished

Now, we can change the label to **Exit** and set the command (just like a regular button widget) to the callback we want PAGE to create for us. In this case, we will use `on_mnuFileExit` as our callback function name.

If you had more options we wanted to add under the **File** menu item, you would click back on **File**, then click on the + icon again. But in this case, we want to add another header item; this time called **Help**. So, click on the top **<Menu>** item, click on the + icon again, and select **New Cascade**. Change the **NewCascade** label item to **Help**. Now, click on the **Help** item and once again click on the + icon and add a **New Command** item. This time, set the label to **About** and the command attribute to **on_mnuHelpAbout**.

To close the Menu Editor, click on the Green Check in the upper right-hand corner of the Menu Editor.

At this point, your Toplevel should look something like this:

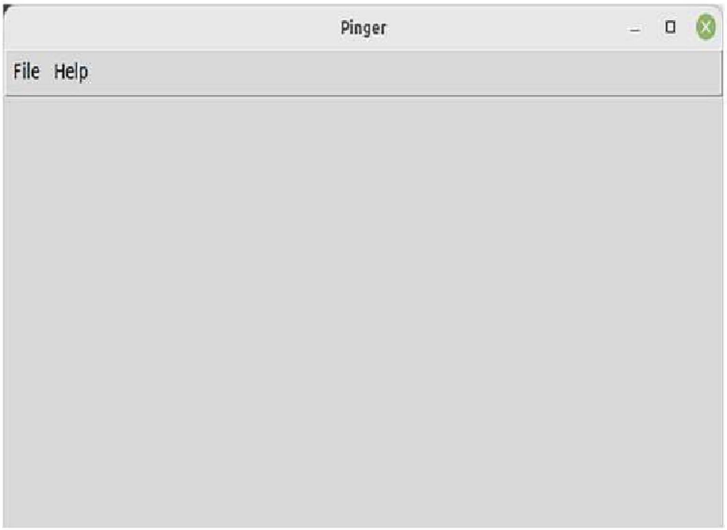


Figure 4.8: Toplevel with Menu Added

Now, we can start the design of the rest of our GUI.

Filling out the rest of the Main GUI

Our main form uses three "standard" Frames and one Labelframe to hold the rest of our widgets and to group them logically.

We'll start with the Frame on the top left. Just below the menu, place a Frame somewhere near the top left of the form. You should move it to location x=4 and y=0. The width should be set to 395 and the height to 175. You should set the borderwidth to 2 and the relief to "groove". Set its attribute to "`frameDisplay`". Next, we will

need to add 10 Label widgets inside this frame. The Labels are positioned into two columns and their size does make a difference. They are set to support the extra languages. When we get to the coding part, I will be explaining the language files, but for now, be sure you are following the width attributes for the Labels as closely as possible. Also, remember that the size of the fonts will make a big difference in the number of characters that can fit in the available space.

The left column of Label widgets all have an x position of 4 and are 147 pixels wide. The y position for the first Label is 15 and each additional Label is set 30 pixels lower. So, for all the y positions, it would be 15, 45, 75, 105, and 135. All have a standard height of 19 pixels. You don't have to set the Alias for any of these Labels. However, you should set the Anchor attribute for all to "`e`" (East) so that the colons (":") will all line up. Text attributes, starting with the top Label, should be: "`External IP:`", "`Last Ping Time:`", "`Pings Done:`", "`Slow Pings:`", and "`Failed Pings:`".

The right column of Labels also line up at x position 155. Each will have a relief attribute of "Sunken" and will be 29 pixels high and have a width of 226. The y position for each should start at 10, so the positions will be 10, 40, 70, 100, and 130. That's 30 pixels spacing, just like the left column. The Alias attributes should be set (starting from the top) to "`lblExternalIP`", "`lblLastPing`", "`lblPingsDone`", "`lblSlowPings`", and "`lblFailedPings`". We set the Alias for these so we can set some of the attributes at run time from our code. Finally, for these Labels, we need to set the `TextVariables` (textvar) Attributes to "`ExternalIP`", "`LastPingTime`", "`PingsDone`", "`SlowPings`", and "`FailedPings`".

The next thing we need to do is place another Frame to hold the Options settings and the Start button. Well, set the alias to this Frame to "`frameOptions`" and should be placed at the x position of 4, y position of 175, width of 395, and height of 165. It should have the relief attribute set to "groove" and a border width of 2. Basically, it should line up with the Frame above.

This Frame will hold 2 Labels, 2 Spinboxes, and 1 Button. Start with the two Label widgets and line them up at the x position of 10. Set the width to 252 and the height to 18. Again, set the y position to 30 pixels apart, which would be 10 and 40. Similar to the left column of Labels in the Frame above, you don't need to set the Alias for these two Labels since they are simply static prompts, but you should set the Anchor alias to "e". The top Label should have the text attribute set to "Delay between Pings (in secs):" and the bottom one to "`Times to ping (0 = forever):`".

Next, place two Tk Spinboxes into the frame, fairly close to the right sides of the two labels we just placed. If you look carefully at the Spinbox widget, you will see that it is a special combination widget made up of an Entry widget with two tiny buttons

stacked on top of each other. While it is designed to work with numerical data, it can work with a list of strings. When you are working with integers or floating numbers, the up arrow will increment the value and the down arrow will decrement the value. If you are working with strings, the up arrow will increment the position in the list and the down arrow will decrement the position in the list. In this case, we will only be working with integers.

The top Spinbox should have an Alias of "**spinPingDelay**" and set the from attribute to 1 and then to attribute to 100. Set the x position to 260 and the y position to 10. The height should be 21 and the width should be 68. Finally for this widget, set the textvar to "**PingDelay**". The second Spinbox should have the alias "**spinTimesToPing**", the from value to **0** and the to value to **10000**. Set the x position to 260 and the y value to 40, the height to 21 and the width to 118. Finally, set the textvar to "**TimesToPing**".

Now, we need to add the Button widget to our frame. Set the Alias to "**btnStart**", the x position to 160, the y position to 100, the height to 37, and the width to 107. Set the text to "**Start**" and the command attribute to "**on_btnStart**".

At this point, before we go any further, save your project.

We are about halfway through our main form and the hardest parts are done. Now, we need to add a Labelframe widget to the right side of our Toplevel form.

We want this to mirror the left two frames in height and width to fill the rest of the space in our form, with a small amount of space on the right side, to duplicate the one on the left. Since this is a Labelframe, you can't get the borders of the frame to line up at the top, since the Label portion of the frame extends beyond the top of the border. Set the Alias to "**lframePingTo**". Set the y position to 0, the x position to 400, the height to 340 and the width to 250. Set the Relief attribute to "**groove**" and the text to "**Ping to Sites**".

Now, we will add five Radiobuttons to our Labelframe. The Aliases should be set to "**rbsite1**", "**rbsite2**", "**rbsite3**", "**rbsite4**", and "**rbsite5**". The anchor alias for all five should be set to "**W**". All five will have the same x position as 20 so they all line up into a nice and neat column. Set the y position for the first one to 40 and increment the y position for each one by 30. That makes the five y positions, 70, 100, 130, and 160. The height for all should be 23 and the width for all should be 215. Now for the somewhat boring part. We still need to set the text attributes, the textvars, and the values. The variable attribute is set by PAGE to a default of "**selectbutton**". Since these five Radiobuttons act as a group, we can leave that as default. Remember, since we are using the five Radiobuttons as a group, the value Attribute for each has to be unique. The top Radiobutton will be set to 0 and each

gets incremented by 1, so that would be 0, 1, 2, 3, and 4. Again, since we are using

these Radiobuttons as a group, we can use the same callback function for all five, so set the command attribute for all five to "`on_rbSiteClick`". Before we set the text and textvar attributes, you need to remember that you need to set them in order, text first then the textvar after that. The reason is the Tk widgets that have a textvar attribute will not allow the label to change once the textvar is set. This is a limitation in the Tk widgets not in PAGE. So, always set the text attribute first, then the textvar afterwards. So, all that having been said, the first Radiobutton should set to an IP address of "8.8.8.8" and the textvar to "`rb1display`". This is the public IP Address of the Google DNS server. Set the second Radiobutton text to "`Google2 - 8.8.4.4`" and the textvar to "`rb2disp`". The final three Radiobuttons can be set to either any other IP address you know or an internal network IP address. If you don't have any other IP address you want to use at this time, you can set the text attributes to " " and the textvar attributes to "`rb3disp`", "`rb4disp`" and "`rb5disp`". I would suggest that if you aren't going to set the text values for any of the remaining Radiobuttons, that you set the state attribute to "`disabled`" so that you can't accidentally click on them when you run the project.

The final thing we need to do is place our final Frame and it is a widget. Place the standard Frame into the lower (blank part) of the Toplevel form. Set its Alias to "`frameTxtBox`", its x position to 4, the y position to 341, the height to 255, and the width to 645. Set the relief attribute to "groove" to match all the other Frames and the borderwidth to 2. Inside this Frame, add a ScrolledText widget. You can find this near the bottom of the Toolbox window. Set the Alias for this widget to "`txtErrors`" so that we can access it later on in the code. Set the x position to 5, the y position to 3, the height to 248 and the width to 636. The reason we are using the ScrolledText widget is to provide a visual indication of any errors showing and the scrollbars make it easy to view the text later on.

That's it for the main form. Save your project and we'll continue.

Adding the About form

The final form for our project will be the About form. We've already touched on creating a multiple-form project back in Chapter 2, so we will jump right in.

Add a new Toplevel form to your project. Set the Alias of this form to "`About`" and the Title to "`About`". Set the width to 459 and the height to 535. Move the form somewhere close to being centered on the screen.

Our About box form has four Label widgets, one Scrolled Text widget and a Button to dismiss the form and return to the Main form. We'll start with the four Labels.

Place the first Label and set its Alias to "`lblProgName`". Set its x position to 10, the y position to 20, its height to 29, and its width to 436. Set the relief to "sunken" and the textvar to "`ProgNameDisplay`". Yes, I know we haven't set the text var yet and for this part of the project, it's ok for it to default to "Label". When we show the form for the first time, we will set the text property for all the Labels from the code before it is shown.

The second Label should have an Alias of "`lblVersion`", the x position of 10, the y position of 52, a height of 29 and a width of 436 (pretty much a duplicate of the first Label). Again, set the relief to "sunken" and the textvar to "`VersionDisplay`". The third Label should have an alias of "`lblAuthor`", x position of 10, y position of 84, height of 29, and width (again) of 436, relief to "sunken" and textvar of "`AuthorDisplay`".

The fourth and final label should have an alias of "`lblCopyright`", the x position of 10, the y position of 1the 16, height of the 29, and the width of 436. Again, set the relief to "sunken" and the textvar to "`lblCopyrightDisplay`".

Now, we will add a ScrolledTextbox to our form and unlike the one in the main form, we won't put it into a frame. You can leave the alias to "`Scrolledtext1`". Set the x position to 10, the y position to 160, height to 291, and width to 438. Now using the Widget Tree window, select the `Scrolledtext` widget, then click on the Text child widget:

Figure 4.9: ScrolledTextbox Child widget Text

We need to do this since we need to set the wrap attribute of the widget to "work", but it doesn't get exposed to the attribute editor for the entire ScrolledText widget. Once you click on the Text widget child, you will see the Attribute Editor change to the child widget and the wrap attribute will be near the bottom just before the Geometry section. It's a dropdown (or combobox), so click on the down arrow of the attribute and then you can set it to "word". This will cause the widget to automatically wrap long lines of text broken at the end of a word.

Finally, we need to place a Button on the form near the bottom. Set its Alias to "`btnAboutDismiss`", its x position to 130, y position to 470, height to 37, and width to 177. Set the text attribute to "Dismiss" and the command attribute to "on

to 177. Set the text attribute to "Dismiss" and the command attribute to "on_
btnAboutDismiss".

That's it. We are done with the GUI design. Save your project and we'll move on to the code.

The code

In many ways, the code for this project is very simple to understand, assuming you have already gotten your "head wrapped around the concept of event-driven" programming. However, some of the code relies on "CLI-based procedural programming, due to the approach we need to take to make the Ping process work properly. However, we will be able to mitigate this issue somewhat. Before we start, we will need to create our language files. These will be needed if you wish to support multiple languages in your version of Pinger.

The language files need to be located in a sub-folder in our project folder called "`languages`". There need to be two files per language that you wish to support. One needs to be called "`data-Strings-{language}.txt`" and the other "`UI-Strings-{language}.txt`". Obviously, the "`{language}`" portion of the filename would be something like "English" or "Spanish" or "Norwegian" or whatever language you wish to use. This can be just about any language with the possible exception of those languages that are read from right to left as opposed to English or Spanish that are read from left to right. Tk widgets don't really support right-to-left read languages, unfortunately. There are ways to "fake it", but for the most part, just assume that they can't be supported.

In each file, there will be a common one-for-one line format between the English file and the other language file. For example, here are the contents of the "data-Strings-English.txt" file:

```
Starting unlimited pings with $delay second delay

Starting $pings pings with $delay second delay

------------------      Finished      ------------------

This program will ping various address to verify that the internet is up.
Delay between pings and the number of pings are able to be set before
ping start.

Copyright

Pinger

G.D. Walters

Debugging turned on - Data will go to log file at $time.
```

```
Python version: $maj.$min.$mic

$time - Log Started

Ping to $to

Local machine name & address: $host - $ip

Local machine name/ip address/gateway: $host - $ip - $gateway

Public IP address: $pubip

$num - $tim - Slow Ping: Min=$min Avg=$avg Max=$max

$num - $tim - External Ping Failed!

Number $num - $tim - Min=$min Avg=$avg Max=$max

Operating Platform: $op
```

There are 19 lines in total for the English file and 18 of those will be duplicated in the target language. Also, notice that various lines have variables defined that start with the "$" symbol. So, now that you understand what the English version of the data-Strings file looks like. Let's take a look at the French version of the same file:

```
Commencer des pings illimités avec $delay second delay

Démarrage de $pings pings avec $delay second delay

------------------     Fini      ------------------

Ce programme ping une adresse différente pour vérifier que l'Internet est en
place. Le délai entre les pings et le nombre de pings peuvent être réglés
avant le démarrage du ping.

droits d'auteur

Pinger

G.D. Walters

Débogage activé - Les données iront au fichier journal à $time.

Version Python: $maj.$min.$mic

$time - Connexion au journal

Ping à $to

Nom de la machine locale et adresse: $host - $ip
```

```
Nom de la machine locale / adresse IP / passerelle: $host - $ip - $gateway

Adresse IP publique: $pubip

$num - $tim - Slow Ping: le minimum=$min moyenne=$avg maximum=$max

$num - $tim - Ping externe a échoué!

Nombre $num - $tim - le minimum=$min moyenne=$avg maximum=$max

Plate-forme d'exploitation: $op
```

As you can see that each line from the English version is duplicated in the French version and the variables (those parts that start with a "`$`") are duplicated as well. I used the Google Translate web page to change the various lines of text.

Similarly, the `UI-Strings-English.txt` file contains the various text attributes that will be displayed in most of the widgets in our project:

```
Pinger

Ping To Sites:

External IP:

Last Ping Time:

Pings Done:

Slow Pings:

Failed Pings:

Delay between pings (in secs):

Times to ping (0 = forever):

Start

Stop
```

Just to show you the difference, we'll look at the contents of the UI-Strings-Norwegian.txt file:

```
Pinger

Ping til nettsteder:

Ekstern IP:

Siste Ping Time:
```

```
Pings ferdig:

Sakte Pings:

Mislyktes Pings:

Forsinkelse mellom Pings (i sekunder):

Tider å pinge (0 = for alltid):

Start

Stoppe
```

I've included five language files in the sample code if you are interested in using the various languages. They are English, French, Italian, Norwegian, and Spanish. If you want to experiment with them, copy the two English files to files of your own, substituting the word "English" in the filename for the target language of your choice. Let's say you want to create language files for German. The filenames would be "`data-Strings-German.txt`" and "`UI-Strings-German.txt`". Then, copy each line from the English version and go to **https://translate.google.com/**. Then, in the left-hand box, paste the line into it, and then select in the right-hand box the target language. Then, copy the output in the right-hand box and paste it into your target language file, making sure that the variables that start with "`$`" are still the same as the English files, since the code and variables are all in English, no matter what your target file is.

I'll try to explain the substitution process in some detail when we get to it, but just for your information, we will use the Template Python Module to do the substitutions.

Now that we've discussed that, we need to open the pinger_support.py module in the IDE or text editor of your choice.

Here is the import section of the support module:

```
import sys

import datetime

import logging

import subprocess

import platform

import os

import requests
```

```
import socket

from string import Template

import locale

# Automatically Provided by PAGE

import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *

# Other imports not provided by PAGE

from tkinter import messagebox

# Another line provided by PAGE

import pinger7
```

Right after the "**import pinger**" line, you need to put in a variable that will ultimately become a global variable:

```
debug = False
```

By placing it here (just under the imports), you don't have to define it in every function you want to use it in. If you were to define it as a global in the startup function, you would need to have a line in every function that would say "**global debug**" before you reference it in your code. This simply saves you a whole bunch of typing.

Now, move down to the main function, which should be just a few lines below. You need to call the **start_up** function just before the **root.mainloop()** line. We've done this before in the first two chapters. Here is what it should look like after your edit. (As I've done before, I'll show the code that you need to either edit or insert in bold and the rest of the PAGE-generated code in "normal" strength:

```
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 = pinger7.pinger(_top1)

    # Creates the About form.

    global _top2, _w2

    _top2 = tk.Toplevel(root)

    _w2 = pinger7.About(_top2)

    start_up()

    root.mainloop()
```

Now, we need to create our **start_up** function. Since this is a function that we need to create from scratch, it will all be in bold font:

```
def start_up():

    # ====================================================

    # function start_up()

    # ====================================================

    # sets important local variables and UI defaults

    # ====================================================

    global Running, NumberOfPings, slow, failed

    global ProgName, Author, Version, TargetIP

    global osplatform, location, temp

    global NumberOfPings, Timer_id, debug, Version

    global copyright_symbol, Language, addresses
```

This section creates a very large number of global variables, all of which we want to define in the **start_up** function. Now that the variables have been declared to be global, we can start to set their values:

```
  copyright_symbol = "\u00A9"

    Version = "0.10.1b"
```

```
location = pinger7._location

osplatform = sys.platform
```

Now, we need to deal with the language files. If you are going to use a language other than English, you need to define it here. In this version, the assumption is that the only languages that will be supported will be English and Norwegian. You need to change the second part of the code to support another language like Spanish, Italian, French, or one that you have defined as a language text ser. The first thing we do is make a system call to determine the local. This call will provide the language that your computer is set up for based on your location. After we get that, we first check to see if the value is "**en**" for English and then set the global variable Language to "English". Otherwise, we check to see if it is "**nn**" for Norwegian. If you don't want to support Norwegian but do want to support another language, get the value that is returned from the **local.setlocale** call and then change this code to support your target language. Then, we make a call to a function called "**do_languages**":

```
# ====================================================

# Setup for multi-language support...

# ====================================================


lang = locale.setlocale(locale.LC_ALL, "")

# print lang[:2]

if lang[:2] == "en":

    Language = "English"

    print("Using English")

elif lang[:2] == "nn":

    Language = "Norwegian"

    print("Bruke norsk")

# Language = 'English'

do_languages(Language)
```

Once the **do_languages** function is done, we come back and continue our **start_up** function. We now call a function called **getPublicIp** and use the .set method for the Label widget to display that information. Then, we clear the rest of the labels in the top portion of the main form:

```
    # Get the public ip and set the label

    _w1.ExternalIP.set(getPublicIp())

    # Clear the empty labels

    _w1.LastPingTime.set("")

    _w1.PingsDone.set("")

    _w1.SlowPings.set("")

    _w1.FailedPings.set("")
```

Now, we will create the list of sites that we want to potentially ping in our program. If you are only using the Google DNS sites, you only need to include the first two. Of course, if you didn't disable the other Radiobuttons, you need to make sure you have a list length that matches how many Radiobutton the user can possibly click on. After that, we create a list holding the aliases of all the Radiobuttons. Finally, for this portion of code, we call the invoke method for Radiobutton #1, which makes the Radiobutton widget think it's been clicked on:

```
    # set defaults

    global siteIpList, rblist

    siteIpList = ["8.8.8.8", "8.8.4.4", "192.168.1.12", "192.168.1.236",
    "192.168.1.67"]

    rblist = [

        _w1.rbsite1,

        _w1.rbsite2,

        _w1.rbsite3,

        _w1.rbsite4,

        _w1.rbsite5,

    ]

    _w1.rbsite1.invoke()
```

Now, we set the text for each of the Radiobuttons programmatically:

```
  TargetIP = siteIpList[0]

    cntr = 0
```

```
    for rb in rblist:
        rb.configure(text=siteIpList[cntr])
        cntr += 1
```

Here, we set the ping delay and the number of times to try to ping the selected site. We also set some more global variable default values:

```
    _w1.PingDelay.set(5)
    _w1.TimesToPing.set(5)
    NumberOfPings = 0
    slow = 0
    failed = 0
    Running = "0"
```

We are getting close to the end of our `start_up` function. We create a string for the copyright label and set some more defaults and finally hide the About form:

```
  global Copyright
    Copyright = "Copyright " + copyright_symbol + " 2018,2022 by G.D.
    Walters"
    temp = []

    print(f"Program Version: {Version}")
    pv = platform.python_version()
    print(f"Running under Python {pv}")
    # Set the path for the language files
    print(f"Working Directory: {location}")

    # Hide the about box for now
    hide_about()
    # centre_screen(652, 605)
```

I have a function called `centre_screen` that I have commented out. If you want to have the main form centered on the monitor, you can uncomment it otherwise, it

will show up where ever you left the main form when you saved the PAGE forms.

The next function is the **read_file** function, which is a support function that simply reads a text file, strips the **Carriage Return Line Feed** (**CRLF**) character, and puts each of the lines into a list that gets returned to the function that called it:

```
def read_file(filename):

    # ====================================================

    # function read_file()

    # ====================================================

    # Read file, strip \n and put lines into a list

    # ====================================================

    with open(filename) as f:

        lines = f.read().splitlines()

    return lines
```

The **getPublicIp** function is another support function that simply requests the website at **https://api.ipify.org**, which returns the public IP address of the machine that visits that address. We'll want that address for our logging functions if anything goes wrong. Your public IP address is different from the local IP address assigned to your computer when you logged into your network, either at home or at work or wherever you are:

```
def getPublicIp():

    # ====================================================

    # function getPublicIp()

    # ====================================================

    # requests public ip address from https://api.ipify.org

    # ====================================================

    try:

        return requests.get("https://api.ipify.org").text

    except:

        return 0
```

The `getLocalInfo` is another support function that will return the machine name, local IP address, and gateway from the network. This works differently depending on the operating system you are using, so we have to make a call the `platform.system` to determine the OS. Once we have that, we check to see whether it is "Windows". If so, we go through the first set of code; otherwise, if it is Mac OS or Linux, we can go through the second part of the code. We make a low-level OS call (courtesy of the Python os library) and grab various information that we will need. If you want to know more about the `os.popen` function, you can find a large amount of information on the web. Use your favorite search engine and look for "`python os.popen`":

```
def getLocalInfo():
    # ====================================================
    # function getLocalInfo()
    # ====================================================
    # gets and sets machine name, local ip address and gateway
    host = socket.gethostname()
    if platform.system() == "Windows":
        # Under windows use 'route print -4', assign to a string.
        # Use .find('Gateway') use that as starting position into string
        # and slice 100 characters. split that. Gateway should be #5
        # ip should be #6
        # ---------------------------------------
        data = os.popen("route print -4").read()
        p = data.find("Gateway")
        s = data[p : p + 100].split()
        # This, however looks WAY cooler
        # s = data[data.find('Gateway'):data.find('Gateway')+100].split()
        gateway = s[5]
        ipaddr = s[6]
    else:
```

```
        gw = os.popen("ip -4 route show").read()
        # slice at 'default via '+12 to ' dev ' = gateway
        # slice at 'src '+4 to 'metric'
        part1 = gw[gw.find("default via ") + 12 : gw.find(" dev ")]
        part2 = gw[gw.find("src ") + 4 :].split()
        gateway = part1
        ipaddr = part2[0]
    return ipaddr, gateway, host
```

Our next function is a long one. It is named **on_tick** and uses the **root.after** Tk function to create a timer that automatically gets called after a defined number of seconds (or milliseconds). We will use this to call the ping system utility in our program. This function relies on the global variables **NumberOfPings**, Running, failed, slow, **TargetIP**, and running. It also writes information to the log file that we will be saving so that we have information if anything goes wrong and we can't reach the site that we are requesting:

```
def on_tick():
    # ====================================================
    # function on_tick()
    # ====================================================
    # emulates a timer to do things on a recurring basis
    # based on root.after routine
    # ====================================================
    global NumberOfPings, Timer_id, Running, failed, slow, TargetIP
    now = datetime.datetime.now().time()
    tim = datetime.datetime.now().strftime("%x %X")
    _w1.LastPingTime.set(tim)
    NumberOfPings += 1
    PingsTotal = int(_w1.TimesToPing.get())
```

In the preceding code, we get the actual time from the datetime module, then we format it into a "human-friendly" format and set the Label `lblLastPing` text to that value. We also increment the number of pings we have done.

In the next block of code, we make a call to another support function called "`do_ping`". When the response comes back, we will check to see if the response is equal to 0. We do this within a try/except block just in case there is an error. Once we have the response, if it is not 0, then we increment the failed variable, set the Label `lblFailedPings` to the value of the failed variable and log the fact that we had a failed ping. In the call to the logging Python library, we also do a substitution to the language we need to use for the output.We also add a line of text to the `ScrolledTextbox`, showing the error, again changing the text variables set up in the Language files.

Finally, we programmatically change the background color for the Label `lblExternalIP` to "red2" to alert the user that there was a failed ping:

```
    try:

        # resp, values = do_ping(PingToAddress, "3", "2")

        resp, values = do_ping(TargetIP, "3", "2")

        if resp != 0:

            failed += 1

            _w1.FailedPings.set(failed)

            logging.warning(LogPingFail.substitute(num=NumberOfPings,
            tim=tim))

            _w1.txtErrors.insert(

                END, LogPingFail.substitute(num=NumberOfPings, tim=tim)
                + "\n"

            )

            print(LogPingFail.substitute(num=NumberOfPings, tim=tim))

            _w1.lblExternalIP.configure(background="red2")
```

Otherwise, we had some success, either the ping was good or it was slow. Again, we set the background color of the Label `lblExternalIP` to "`green2`" and if the ping response time (returned in the call to `do_ping` in the values variable), then we reset the color to "yellow". If the ping response was slow, we also log the fact as a warning

and add the information to the ScrolledTextbox. If it was not slow, then we log the fact that it was a good ping. We'll see more about the logging function later on in the code:

```
        else:
            _w1.lblExternalIP.configure(background="green2")
            if float(values[1]) > 50.0:
                slow += 1
                _w1.SlowPings.set(slow)
                _w1.lblExternalIP.configure(background="yellow")
                logging.warning(
                    SlowPingDisplay.substitute(
                        num=NumberOfPings,
                        tim=tim,
                        max=values[2],
                        avg=values[1],
                        min=values[0],
                    )
                )
```

This block of code will add the fact that we had a slow ping to the Scrolled Textbox widget:

```
                _w1.txtErrors.insert(
                    END,
                    SlowPingDisplay.substitute(
                        num=NumberOfPings,
                        tim=tim,
                        max=values[2],
                        avg=values[1],
                        min=values[0],
```

```
            )
            + "\n",
        )
        print(
            SlowPingDisplay.substitute(
                num=NumberOfPings,
                tim=tim,
                max=values[2],
                avg=values[1],
                min=values[0],
            )
        )
```

Otherwise, we consider it good and log the fact along with the data returned from Ping:

```
    else:
        logging.info(
            LogPingGood.substitute(
                num=NumberOfPings,
                tim=tim,
                min=values[0],
                avg=values[1],
                max=values[2],
            )
        )
        print(
            LogPingGood.substitute(
                num=NumberOfPings,
```

```
                tim=tim,
                min=values[0],
                avg=values[1],
                max=values[2],
            )
        )
```

Finally, if we set the debug variable (or flag if you will), to True just after our import section, we also print the good information to the ScrolledTextbox:

```
        if debug is True:
            _w1.txtErrors.insert(
                END,
                LogPingGood.substitute(
                    num=NumberOfPings,
                    tim=tim,
                    min=values[0],
                    avg=values[1],
                    max=values[2],
                )
                + "\n",
            )
```

Now, we are at the except part of our try/except block. This section will handle any other errors that we haven't accounted for in our code above. Here, we set the Label **lblExtermalIP** background to "**red2**", increment the failed variable count, set that value in **lblFailedPings**, send that information to the log file, add to the ScrolledTextbox, and if the debug flag is set to True, print:

```
    except:
        _w1.lblExternalIP.configure(background="red2")
        failed += 1
```

```
        _w1.FailedPings.set(failed)

        logging.warning(LogPingFail.substitute(num=NumberOfPings,
        tim=tim))

        _w1.txtErrors.insert(

          END, LogPingFail.substitute(num=NumberOfPings, tim=tim) + "\n"

        )


        if debug:

            print(LogPingFail.substitute(num=NumberOfPings, tim=tim))
```

Now that we are done with the ping, at least for this one, we can finally update the **lblPingsDone** Label and check to see whether we have completed the requested number of pings. If so, we update the log and the ScrolledTextbox. We also set the text of the button (**btnStart**) back to "Start" (which is set to "Stop" in the **on_btnStart callback** function) and cancel the timer with **root.after_cancel**. Otherwise, we set the **root.after** that to the next needed number of seconds to delay:

```
    _w1.PingsDone.set(NumberOfPings)

    if int(_w1.TimesToPing.get()) != 0:

        if NumberOfPings >= int(_w1.TimesToPing.get()):

            print(LogFinish)

            root.after_cancel(Timer_id)

            Running = "0"

            logging.info(LogFinish)

            _w1.txtErrors.insert(END, LogFinish + "\n")

            _w1.btnStart.config(text="Start")

        else:

            delay = int(_w1.PingDelay.get()) * 1000

            Timer_id = root.after(delay, on_tick)

    else:

        delay = int(_w1.PingDelay.get()) * 1000
```

```
        Timer_id = root.after(delay, on_tick)
```

The function `do_ping()` is next up. Three parameters are passed into it: address, count, and wait. As you can guess, the address is the IP address that we are trying to ping, the number of times to ping in this call and the number of seconds to wait for the child process to finish. It returns a response value and (if successful) a list containing the minimum time for the round trip, the average time of all the pings, the maximum amount of time and mdev, or the standard deviation of the ping times (which in many instances is not really need unless you are having issues with VOIP or very large bulk transfers). In Windows, mdev is not available so only the first three values are returned. If the ping was not successful, then we return whatever error code we got as the response (0 if it was good) and then a list containing 4 zeros ([0,0,0,0]).

Since we are doing low-level system calls again, we must check, whether the OS is Windows, Mac, or Linux. Luckily, the only one that needs special attention is Windows. If you are running on Mac or Linux, it would be a good idea to look at the Windows part of the code as well and the bottom section, since they are both fairly similar:

```
def do_ping(address, count, wait):

    # ================================================
    # function: do_ping()
    # ================================================
    # Expects:
    #    address - String - ip address of the site to ping
    #    count   - String - Number of times to ping in this call
    #    wait    - String - Number of seconds to wait for the
    #                       child process to terminate
    # --------------------------------
    # Returns:
    #    On good : 0, [min time, avg time, max time, mdev time]
    #    On error: response, [0,0,0,0]
    # --------------------------------
```

```
print(
    f"Attempting to ping IP {address} {count} times with a delay of
    {wait} between each attempt"
)
```

So, we check the value of **platform.system** again to see if we are running under Windows. If so, we use this part of the code block. Not only does Windows use a different command line structure for ping, but it also returns the values in a different order (and does not return mdev) than a Linux or MacOS machine. We use the **subprocess.Popen** call to call the ping utility, which allows you to create a new process for that utility and connect to the processes input, output, and error pipes and get any error codes. For more information on the Python subprocess module, you can do a web search, or go to the Python documentation site. This call can become extremely complicated very quickly:

```
if platform.system() == "Windows":
    # Windows version of ping uses a different command for
    # count and wait than the Linux version does...
    p = subprocess.Popen(
        ["ping", address, "-n", count, "-w", wait],
        stdout=subprocess.PIPE
    )
    tempresp = p.wait()
    pcom1 = p.communicate()
    pcom = pcom1[0]
    if tempresp == 0:
        resp = 0
    else:
        resp = 1  # pcom1[1]
```

In the preceding code block, we use the **p.wait()** method to wait for the "child" process (our ping command) to finish or terminate. The return value (**tempresp**) holds the fact that the process has finished if it's a 0 or a None if the process hasn't finished. The **p.communicate()** method returns a tuple of **stdout_data** and **stderr_**

`data`. This data will be in byte format. Since the data returned will be in byte format, we need to use the .decode method to convert it back to "standard" strings. We then can manipulate the string (s) through a series of calls to arrive at a string that we can split into three different values, pack them into a list, and return them along with the general response of 0 or 1 based on if the call succeeded or not:

```
        if resp == 0:
            # It also returns the data in a different format than
            # the linux version does.

            s = pcom.decode("ascii")
            s1 = s[s.find("Minimum = ") :]
            s1 = s1.replace("ms", "")
            s1 = s1.replace(",", "")
            s2 = s1.split()
            min = s2[2]
            max = s2[5]
            avg = s2[8]
            values = [min, avg, max]
            t = values
        else:
            t = 1
            values = [0, 0, 0]
        return resp, t
```

This part of the code block is for Mac and Linux machines. It's pretty much the same thing as the Windows code with a very few exceptions. Notice that under Windows, we needed to call `p.wait()` to wait for the subprocess to end. This is not needed outside of Windows. When the communication method is finished, we get the return information in a bit of a different format than the Windows machine returns. In the case of Mac and Linux, we have to look at the second part of the tuple to see if the call was successful (None or an error code). The first part of the tuple contains the actual data we want to use to show the times. As in Windows, we need to decode the

byte stream. Then, we can manipulate the data to find our values, pack them into a list, and return them:

```
    else:
        try:
            p = subprocess.Popen(
                ["ping", address, "-c", count, "-W", wait],
                stdout=subprocess.PIPE
            )

            tmp = p.communicate()
            pcom = tmp[0].decode("ascii")
            if tmp[1] is None:
                resp = 0
            else:
                resp = tmp[1]
            start = pcom.find("rtt min/avg/max/mdev = ")
            end = pcom.rfind("ms")
            values = pcom[start + 23 : end]
            t = values.split("/")
        except:
            resp = 1
            t = [0, 0, 0, 0]
        return resp, t
```

So that is the `do_ping` function wrapped up. That was all of the low-level stuff, so now we can concentrate on more "usable in a real-world" helper packages. We'll concentrate on the logging portion for now (there's more to come). Logging can be very helpful to programs you create on your own. Take a look at a portion of a log from a successful run:

```
INFO : Python version: 3.7.4

INFO : Operating Platform: linux

INFO : Pinger Version: 0.10.1b

INFO : ================================================

INFO : 09/04/2022 10:41:06 AM - Log Started

INFO : Ping to 8.8.8.8

INFO : Local machine name/ip address/gateway: earth - 192.168.1.81 -
192.168.1.1

DEBUG : Starting new HTTPS connection (1): api.ipify.org:443

DEBUG : https://api.ipify.org:443 "GET / HTTP/1.1" 200 12

INFO : Public IP address: 67.10.205.40

INFO : Starting 5 pings with 5 second delay

INFO : Number 1 - 09/04/2022 10:41:06 AM - Min=18.272 Avg=19.094 Max=20.057
```

I like to provide (for most of my serious programs) data on the version of Python that was running the code (which can be a big help if there are version errors), the OS that the program was run on and the current version of the program. Notice that each line in this log starts with either "INFO :" or "DEBUG :". This gives some important data by itself since we can see just at a glance that there were no errors or warnings (at least in this snippet of the log). There are six levels of logging entries. They are "CRITICAL", "ERROR", "WARNING", "INFO", "DEBUG", and "NOTSET". I'm sure you can begin to see the amount of help a log can provide, especially one that is very easy to implement:

```
def start_log():

    # ====================================================

    # function: start_log

    # ====================================================

    # initiates the log file and enters base information...

    # ====================================================

    global siteIpList, rblist

    ver = sys.version_info
```

```
IP, Gateway, hostname = getLocalInfo()

now = datetime.datetime.now().time()

tim = datetime.datetime.now().strftime("%x %X")

logging.basicConfig(

    filename="Pinger.log",

    filemode="w",

    format="%(levelname)s : %(message)s",

    level=logging.DEBUG,

)
```

We call the `start_log()` once the "Start" button is clicked. Creating a log file from scratch can be difficult, especially if you are dealing with difficult data. The line `logging.basicConfig` does that for us. We simply pass in the filename (including the path if you need to write to somewhere not in the current folder), and the file mode (overwrite, append, and so on.). Once this is done, we simply call the function that you want to write data to at the level that you wish. For example, the next line writes an INFO line (below) to the log that provides the Python version used to run the program. The line we are going to print to the log is the ninth line of the **data-Strings-English** (or French or Spanish or Insert Language file here) file. It looks like this:

```
Python version: $maj.$min.$mic
```

When it was read in from the file (which we haven't gotten to yet) the string is named PyVer and when we call `PyVer.substitute()`, the variables within the string (those that start with "$" are replaced with the proper values. We'll cover that in a little bit. I just wanted you to have a gross grasp of what was happening in the next lines as we print them to the log and to the terminal window if we have to debug set to True:

```
logging.info(PyVer.substitute(maj=ver[0], min=ver[1], mic=ver[2]))

if debug:

    print(PyVer.substitute(maj=ver[0], min=ver[1], mic=ver[2]))

# logging.info("Operating platform: {0}".format(osplatform))


logging.info(OpPlatform.substitute(op=osplatform))
```

```
if debug:
    print(OpPlatform.substitute(op=osplatform))
logging.info("Pinger Version: {0}".format(Version))
logging.info("================================================")
logging.info(LogStart.substitute(time=tim))

if debug:
    print(LogStart.substitute(time=tim))
```

Now, we call the get method for the Radiobuttons variable (`selectbutton`) to find out which button was set when the program was started and the "Start" button was clicked. Remember the Radiobutton value attribute for each button was set at 0 for the first Radiobutton, 1 for the next, and so on. This gets set to the which variable. We then use that value to look at the siteIpList as an index into the list of IP addresses:

```
which = _w1.selectbutton.get()
# which = siteIpList[siteiplist]
pingto = siteIpList[which]
if debug:
    print(f"Target IP: {pingto}")
logging.info(PingToDisp.substitute(to=pingto))
if debug:
    print(PingToDisp.substitute(to=pingto))
logging.info(
    LocalNameAddressGateway.substitute(host=hostname, ip=IP,
    gateway=Gateway)
)
if debug:
    print(LocalNameAddressGateway.substitute(host=hostname, ip=IP,
    gateway=Gateway))
```

```
    # if you are having issues with your internet connection, you
    # might have to 'force' the PublicIP address.
    pubip = getPublicIp()
    if pubip != 0:
        if debug:
            print(PublicIP.substitute(pubip=pubip))
        logging.info(PublicIP.substitute(pubip=pubip))
    else:
        if debug:
            print(PublicIP.substitute(pubip=_w1.ExternalIP.get()))
        logging.info(PublicIP.substitute(pubip=_w1.ExternalIP.get()))
```

The rest of the function just prints the beginning information to the log file. Now, we'll look at the callback function for the "`Start`" button. Remember, this function is called whenever the button is clicked. Therefore, we have to handle both a "`Start`" and "`Stop`" situation:

```
def OnBtnStart():
    # ====================================================
    # callback function OnBtnStart()
    # ====================================================
    # Called when the 'Start' button on UI is clicked
    # ====================================================
    if debug:
        print("PingerML_support.OnBtnStart")
        sys.stdout.flush()
    global NumberOfPings, Running, Timer_id
    global failed, slow
    start_log()
    # ---------------------------------
```

You can see the call to the `start_log()` function that we just got through the discussion. Now, we look at the global variable "Running". This is set to either a 0 or a 1. We can use this simple value to see if we have started running a process. If it is a 0, then we haven't started yet, so we need to start. If it's a 1, then we have started and we need to stop the process and clean up after ourselves.

To start the ping process, we clear some variables, set the information into some of the labels Labels on our main form, set the "Running" global to 1, change the text on the button from "Start" to "Stop", write some more information into the Log file, and finally, call the `root.after()` method to start the timer. When we do this, we provide the time to wait until the process is actually called, which in this case is 0, which means immediately and the callback function that we want to use. In this case, it is the `on_tick` function that we discussed a little bit ago. We receive a timer ID that we need to keep in order to stop the process properly:

```
    if Running == "0":
        NumberOfPings = 0
        failed = 0
        slow = 0
        _w1.FailedPings.set(failed)
        _w1.SlowPings.set(slow)
        Running = "1"
        _w1.btnStart.config(text=StopButtonLabel)
        delay = _w1.PingDelay.get()
        if debug:
            if _w1.TimesToPing.get() == 0:
                logging.info(LogLine1.substitute(delay=delay))
            else:
                logging.info(
                    LogLine2.substitute(pings=_w1.TimesToPing.get(),
                    delay=delay)
                )
        Timer_id = root.after(0, on_tick)
```

```
    else:

        Running = "0"

        logging.info(LogFinish)

        _w1.txtErrors.insert(END, LogFinish + "\n")

        _w1.btnStart.config(text=StartButtonLabel)

        root.after_cancel(Timer_id)  # Cancel the 'timer'
```

Now, we can get to the case where we are running (the global Running variable is set to 1). First, we set Running back to 0, we write the log that we are finishing the run, write that also to the ScrolledTextbox widget on our main form, set the text of the Start button back to "Start", and then call `root.after_cancel()` method with the timer ID of the timer to stop the timer process. That was easy!

Now, we will take a look at the `do_languages` function. It gets passed a language name like "English" or "Norwegian" and it will then load in the proper language files, both the data file (for printing to the log and terminal window) and the UI file (which is for the GUI widgets. There is nothing complicated here. We use an if/elif tree to decide which files to have read, based on the language string passed into this function, the sets the variables `uiFilename` and `dataFilename`. Once we fall out of the if/elif tree, we call the next functions to read in the a from the appropriate text file and load the strings into the main form widgets:

```
def do_languages(Language):

    # ====================================================

    # function do_languages()

    # ====================================================

    # assigns the proper language files based on variable Language

    # ====================================================

    global location

    print(f"Base Location: {location}")

    uiFilename = ""

    basedir = os.path.join(location, "languages")

    if Language == "English":
```

```
        uiFilename = os.path.join(basedir, "UI-Strings-English.txt")
        dataFilename = os.path.join(basedir, "data-Strings-English.txt")
    elif Language == "Norwegian":
        uiFilename = os.path.join(basedir, "UI-Strings-Norwegian.txt")
      dataFilename = os.path.join(basedir, "data-Strings-Norwegian.txt")
    elif Language == "Spanish":
        uiFilename = os.path.join(basedir, "UI-Strings-Spanish.txt")
        dataFilename = os.path.join(basedir, "data-Strings-Spanish.txt")
    elif Language == "French":
        uiFilename = os.path.join(basedir, "UI-Strings-French.txt")
        dataFilename = os.path.join(basedir, "data-Strings-French.txt")
    elif Language == "Italian":
        uiFilename = os.path.join(basedir, "UI-Strings-Italian.txt")
        dataFilename = os.path.join(basedir, "data-Strings-Italian.txt")
    else:
        print("FAIL")
        uiFilename = os.path.join(basedir, "UI-Strings-English.txt")
        dataFilename = os.path.join(basedir, "data-Strings-English.txt")
    if debug:
        print(f"UI Filename: {uiFilename}")
        print(f"Data Filename: {dataFilename}")
    load_UI_strings(uiFilename)
    set_UI_strings()
    load_Data_strings(dataFilename)
```

At this point, we get to see how the strings for the GUI main form and the logging get to read from the file and converted into the various variables. We'll start with the UI strings. There are 11 lines that make up the UI file. When the file is read, each line is read into a variable named data. Each line is there for a position within

that variable. Line 1 is the name of the program. This translates to data[0] and gets assigned to the global variable **ProgName**. So on and so forth. With this knowledge, you should be able to understand the rest of the function:

```
def load_UI_strings(uiFilename):

    # ====================================================

    # function load_UI_strings()

    # ====================================================

    # Loads the UI-Strings-Language file from the list

    # created by read_file() and assigns each item to

    # a global variable to load UI static information.

    # ====================================================

    global ProgInfo, Copyright, ProgName, AuthorName, DebugOn, PyVer,
    LogStart, Version

    # Read the sample file

    data = read_file(uiFilename)

    # There should have been 11 lines loaded from the sample file. (0-10)

    global ProgName

    ProgName = data[0]


    global SitesLabel

    SitesLabel = data[1]

    global ExternalIPLabel

    ExternalIPLabel = data[2]

    global LastPingTimeLabel

    LastPingTimeLabel = data[3]

    global PingsDoneLabel

    PingsDoneLabel = data[4]

    global SlowPingsLabel
```

```
    SlowPingsLabel = data[5]
    global FailedPingsLabel
    FailedPingsLabel = data[6]
    global PingDelayLabel
    PingDelayLabel = data[7]
    global TimesToPingLabel
    TimesToPingLabel = data[8]
    global StartButtonLabel
    StartButtonLabel = data[9]
    global StopButtonLabel
    StopButtonLabel = data[10]
```

Now, we'll take a look at the function `set_UI_strings()`. As the name suggests, we are taking the data from the file and variables we just loaded and setting the proper text for the various widgets. Again, you should be able to make sense of this at this point:

```
def set_UI_strings():
    # ====================================================
    # function set_UI_strings()
    # ====================================================
    # after globals are set in load_Data_strings function
    # this routine assigns those values to static text on UI
    # ====================================================
    global ProgName, SitesLabel, ExternalIPLabel, LastPingTimeLabel
    global PingsDoneLabel, SlowPingsLabel, PingDelayLabel
    global TimesToPingLabel, StartButtonLabel, StopButtonLabel
    root.title(ProgName + " " + Language)  # Title
    _w1.lframePingTo.config(text=SitesLabel)  # Ping to sites
    _w1.Label1.config(font=("DejaVu Sans", 9))
```

```
    _w1.Label1.config(text=ExternalIPLabel)  # external IP
    _w1.Label3.config(font=("DejaVu Sans", 9))
    _w1.Label3.config(text=LastPingTimeLabel)  # last ping time
    _w1.Label5.config(font=("DejaVu Sans", 9))
    _w1.Label5.config(text=PingsDoneLabel)  # pings done
    _w1.Label7.config(font=("DejaVu Sans", 9))
    _w1.Label7.config(text=SlowPingsLabel)  # slow pings
    _w1.Label9.config(font=("DejaVu Sans", 9))
    _w1.Label9.config(text=FailedPingsLabel)  # failed pings
    _w1.Label11.config(font=("DejaVu Sans", 8))
    _w1.Label11.config(text=PingDelayLabel)  # delay between
    _w1.Label12.config(font=("DejaVu Sans", 8))
    _w1.Label12.config(text=TimesToPingLabel)  # times to ping
    _w1.btnStart.config(text=StartButtonLabel)  # Start button
```

Next is the `load_Data_strings` function. Again, we pass the filename of the file holding the data language strings and "maps" them to various global variables. All we have to do is pass the filename as a parameter:

```
def load_Data_strings(Filename):
    # ====================================================
    # function load_data_strings()
    # ====================================================
    # Loads the data-Strings-{Language} file from the list
    # created by read_file() and assigns each item to
    # a global variable
    # ====================================================
    data = read_file(Filename)
    # There should have been 18 lines loaded from the sample file. (0-17)
```

```
global LogLine1

LogLine1 = Template(data[0])

global LogLine2

LogLine2 = Template(data[1])

global LogFinish

LogFinish = data[2]

global ProgInfo

ProgInfo = data[3]

global Copyright

Copyright = data[4]

global ProgName

ProgName = data[5]

global AuthorName

AuthorName = data[6]

global DebugOn

DebugOn = Template(data[7])

global PyVer

PyVer = Template(data[8])

global LogStart

LogStart = Template(data[9])

global PingToDisp

PingToDisp = Template(data[10])

global LocalNameAddress

LocalNameAddress = Template(data[11])

global LocalNameAddressGateway

LocalNameAddressGateway = Template(data[12])

global PublicIP
```

```
    PublicIP = Template(data[13])

    global SlowPingDisplay

    SlowPingDisplay = Template(data[14])

    global LogPingFail

    LogPingFail = Template(data[15])

    global LogPingGood

    LogPingGood = Template(data[16])

    global OpPlatform

    OpPlatform = Template(data[17])
```

All these variables now contain the information to support the substitution as we saw in the `start_log()` function. Remember the line: `logging.info(PyVer.substitute(maj=ver[0], min=ver[1], mic=ver[2]))`

which printed the Python version to the logging file. This is where the variable PyVer got set initially and then the substitute function from the Template library module.

When we designed the menu for our main form, we told them that there was a callback command for the Exit menu item and one for the **Help|About** menu item. We will deal with those callbacks that PAGE created for us now. First, the mnuFileExit callback exits the program. You've seen this before in *Chapter 1, Introduction to PAGE and Chapter 2, Going Further*, so we won't spend any more time on it:

```
def mnuFileExit():

    # ====================================================

    # function mnuFileExit()

    # ====================================================

    # called when menu item File | Exit is clicked

    # ====================================================

    if debug:

        print("PingerML_support.mnuFileExit")

        sys.stdout.flush()

    sys.exit()
```

The callback function mnuHelpAbout is almost as simple as the exit function. This function will hide the main form and show the About form. However, in between the two actions, we need to fill the text into the Label widgets as well as the ScrolledTextbox:

```
def mnuHelpAbout(*args):

    # ====================================================

    # function mnuHelpAbout()

    # ====================================================

    # called when menu item Help | About is clicked

    # ====================================================

    # Show the generic about box

    # ====================================================

    if debug:

        print("PingerML_support.mnuHelpAbout")

        sys.stdout.flush()

    # ====================================================

    # Define the globals and load the information into the Labels in the
    form

    # ====================================================

    global temp

    global ProgInfo, Copyright, ProgName, AuthorName

    global DebugOn, PyVer, LogStart, Version

    hide_main()

    copyright_symbol = "\u00A9"

    _w2.ProgNameDisplay.set(ProgName)

    _w2.AuthorDisplay.set(f"Written by {AuthorName}")

    _w2.VersionDisplay.set(f"Version {Version}")

    _w2.CopyrightDisplay.set(Copyright)
```

```
    _w2.Scrolledtext1.insert(END, ProgInfo)

    root.update

    show_about()
```

As you can see, we define the global variables we need to deal with, assign the special character for the copyright symbol (©), and then for the labels, we use the set method of the proper Label widgets. The `ScrolledText` box is a bit different, but we simply use the insert method for the `ScrolledText` widget, with the parameters "END" which places the insertion cursor at the end of the last text (which so far there hasn't been any, so it places it at the top of the first line of the text widget) and then the data we want to put in, which is the program information. By the way, this is the same way to insert text into the "standard" Tk Text box widget. It's just nice to have one that has Scrollbars already set up for us, which is why we used the `ScrolledTextbox` widget.

Now, we have the `centre_screen` function, which simply centers the form in the center of the screen:

```
def centre_screen(wid, hei):

    ws = root.winfo_screenwidth()

    hs = root.winfo_screenheight()

    x = (ws / 2) - (wid / 2)

    y = (hs / 2) - (hei / 2)

    root.geometry("%dx%d+%d+%d" % (wid, hei, x, y))
```

We have used functions to show and hide forms back in *Chapter 2, Going Further*, so we will not spend any extra time on something you have seen before:

```
def show_main():

    global _top1

    _top1.deiconify()


def hide_main():

    global _top1

    _top1.withdraw()
```

```
def show_about():
    global _top2
    _top2.deiconify()

def hide_about():
    global _top2
    _top2.withdraw()
```

We are down to the last two functions in our code. First is the `on_rbSiteClick` callback for the Radiobutton widgets. By this time, it should be very easy for you to understand it. We get the value of the Radiobutton that was clicked using the `selectbutton.get` function, then use that value (zero-based) as the index into the list we created earlier that holds the strings representing the IP addresses of the sites that the program can ping:

```
def on_rbSiteClick(*args):
    if debug:
        print("pinger7_support.on_rbSiteClick")
        for arg in args:
            print("    another arg:", arg)
        sys.stdout.flush()
    which = _w1.selectbutton.get()
    global siteIpList, TargetIP
    # global PingToAddress
    TargetIP = siteIpList[which]
    print(f"PingTo: {TargetIP}")
```

Finally, we have the callback for the dismiss button on the About form. Again, you should be able to write this down, without seeing it, but just in case, here it is:

```
def on_btnAboutDismiss(*args):
    print("pinger7_support.on_btnAboutDismiss")
    for arg in args:
```

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

    hide_about()

    show_main()
```

That is the entire program. All the code, including the PAGE forms and the languages is already created for you and available in the sample code.

Conclusion

Compared to the first three chapters, you learned a huge amount in this chapter. Just as a reminder, here are some of the things you have learned:

- Creating a program that easily supports multiple languages.
- Using Python to control an operating system utility.
- Creating Log files.
- Creating strings from a text file
- Substituting data in Strings using templates.
- Controlling the colors of various widgets on the fly.

While we only touched on things like the subprocess calls, the template used to change variables within a string and using the logging module, you have seen the power of these Python modules to make your programs more professional and easier to debug if there are problems. We have only begun. There is a lot more to come!

I am sure you are "*chomping at the bit*" to get onto the next chapter, but you should take a little time for yourself and rest. Learning is a wonderful thing, but too much learning at one time can get things jumbled.

In our next chapter, we will look at using graphics to enhance the GUIs we produce.

Multiple Choice Questions

1. What is the special character that you use when creating a string with embedded variables for the Template module?
   a. *
   b. $

c. %

2. When using a Radiobutton which attribute do you use to define the variable that is used for knowing which Radiobutton?
   a. value
   b. textvar
   c. Alias
3. How do you set the background color of a Label widget?
   a. set the image attribute to a graphic file.
   b. use the following command {widgetname}.configure(foreground={colorname}
   c. use the following command {widgetname}.configure(background={colorname}

Answers

1. B – Use the $variablename syntax.
2. A – You should use the value attribute
3. C – Use {widgetname}.configure(background={colorname}

Questions

1. Can you use template strings with code or do you have to read it from a file?
2. What is the command to create a log entry at the warning level?
3. How do you change the text on a Button widget in code?

Answer to the questions

1. YES! The template string can be used directly from the code. You do not have to create a file to hold the strings. Just create the string as you normally would. Just remember to use the `${variablename}` format.
2. To create a log entry at the warning level, you would use the following command:

```
logging.error("Your desired error message here")
```

3. There are two ways to set the text attribute of a Button widget:
   - use the text attribute along with the configure command

```
w1.Button1.configure(text="Your text here")
```

_w1.Button1.configure(text="Your text here")

- use the textvar attribute directly and the set method (much easier)
  _w1.textvarname.set("Your text here")

Key terms

- background color
- foreground color
- logging
- template
- subprocess

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5
Using Graphics

Introduction

In our previous chapters, we haven't really had any reason to include images or graphics in our **Graphical User Interface** (**GUI**) forms. In this chapter, we will learn the widgets that can handle graphic images, how to include graphic files on your widgets, how to create your own images, and where to place your images for the easiest use.

It is important for all PAGE users, but especially for those who run Windows as their OS, to start PAGE properly. When starting PAGE, open a terminal window in the project folder and start PAGE from the command prompt or command line terminal. This makes sure that PAGE has the proper path reference for dealing with your graphics. Also, rather than keeping your graphics in your main project folder, keep them in a sub-folder within your main project folder.

Name that folder something like "images", "graphics", "icons" or "assets". These

Name that folder something like "images", "graphics", "icons" or "assets". These few small steps will make it much easier for you to create and maintain your GUI projects, especially those that use graphics.

What you will learn

In this chapter, we will learn the following topics:

- Where to get quality graphics
- Converting image formats
- Creating the GUI
- Writing the Code
- How to embed graphic images into your project with the PAGE designer
- How to load graphic images into your project from the code
- The proper place to place your graphics

Before you start

Before you start your GUI, you should already have all the graphics that you want to use in your project in the project images folder. Yes, you can grab other graphics anytime during the design process, but it would be much easier and feel less of a rush if you already have your graphics ready to go.

Make sure your graphics are of the correct size. If you are making a button bar, you should consider either 32x32 pixels or 46x46 pixels depending on how many buttons you need on your button bar.

Make sure that your graphics are all in the correct format. Tk widgets only support certain graphic formats. These are:

- .xbm format
- .gif format
- .pgm format
- .ppm format
- .png format

Out of all of these formats, the most useful format is the .png format. That is pretty

much a "universal" standard for quality graphics.

As of PAGE 7.0, you can also use .jpg format images. However, there are some drawbacks while using .jpg images. Most importantly, the fact that only .png and .gif format images support a transparent background, which means you will be much happier with the end product. It is simple enough to convert a .jpg to .png or even .svg format images to .png. You can use programs like **Adobe Photoshop** or **The Gimp** to convert one image format to another. Another good program is

"ImageMagik" which is available for Linux, Windows, and MacOS X. Most probably, it is free for most operating systems and can read and write over 200 image formats including .png, .jpg, .gif, .webP and .svg to name a few.

You can find more information at **https://imagemagick.org/index.php**.

There are only a few widgets that accept graphic images. There are a few that support images, but only in certain ways. The *"normal"* graphic supporting widgets are the **Menu**, **tk::Button**, **tk::Label**, **ttk::Tbutton,** and the **ttk::TLabel**. You can also use images in the **tk::Checkbutton**, **tk::Radiobutton**, **ttk::TCheckbutton**, **ttk::TRadiobutton**, and the **ttk::Treeview**.

We will take a look at them, using the tk::Checkbutton and the tk::Radiobutton when we create our GUI. The use of graphics in the ttk::TCheckbutton and the ttk:TRadiobutton is very complicated and it is not advisable to use these widgets when you need to use special graphics. We will talk about the ttk::Treeview in an upcoming chapter.

Getting quality graphics

If you are lucky enough to run Linux as your operating system, there is a wealth of icons in the *usr* folder, a sub-folder called icons. Its actual location is /usr/share/icons. Since MOST Linux distributions are free and open source, these icons are available for you to use in your own projects. Another source of great graphics is **https://openclipart.org**. Almost everything on that site is free and available to use in your projects, many without having to provide attribution (the site name and artist's name). However, in your code and somewhere in your documentation for your program, be sure to mention the site (or OS source) as well as the artist's name if you can find it, even if it is listed as anonymous.

Please do not steal resources.

There are many other sites on the Internet that provide quality graphics; many of them for free. Entire books can be created on all the resources on the web that would be bigger than this one with all the resources out there. Not to mention as soon as it

be bigger than this one with all the resources out there. Not to mention as soon as it got published, it would be out of date with new sites starting up and shutting down.

Converting image formats

When you will need to create or convert images from one format to another, use Gimp. It is a free program that is available for every operating system that PAGE can run on. Many users think that it is as powerful and capable as **Adobe Photoshop**; it is free and available on Linux as well as Mac and Windows.

Since the actual process of creating graphics is off-topic for this book, so I will direct you to search the Internet for beginner guides.

Creating the GUI

We will start off, as always, with a new project. We will call it **graphics**. Here is what it will look like when we finish:

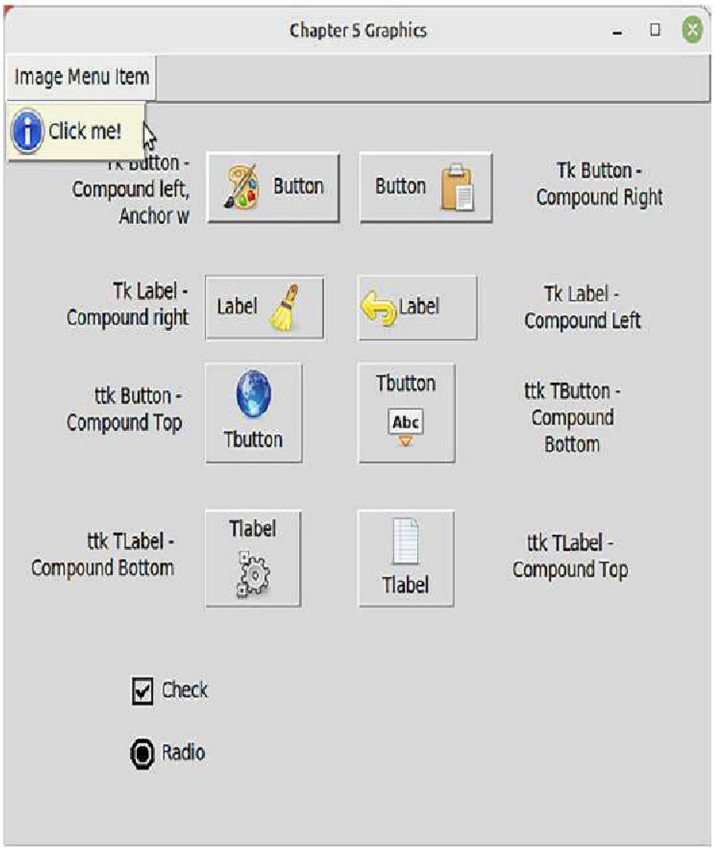


Figure 5.1: Completed Project

Once you have your project setup, set the Toplevel size to 600 pixels wide and 474 pixels high. Move it to somewhere near the middle of your screen and then set the title attribute to "*Chapter 5, Graphics*".

The next thing that we will do is create a menu. We did that in our previous chapter, but we will go over the steps again just to be safe.

In the Attribute Editor, find the menu attribute and click on the part that says "Click":

Figure 5.2: Menu Editor Launch in the Attribute Editor

That will bring up the menu editor, as shown in *Figure 5.3*:

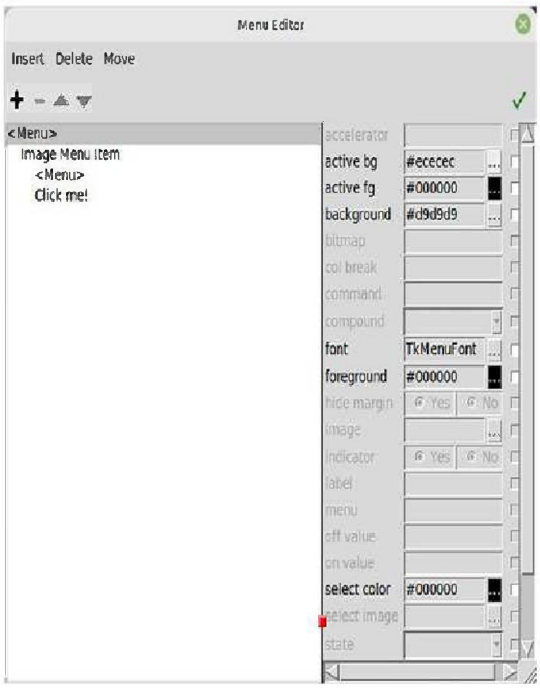


Figure 5.3: Menu Editor

Add a Cascade Item and set the text to `Image Menu Item`. Then, add a Command Item and set the text to `Click me!` and the command to "`on_mnu_Click_me`":

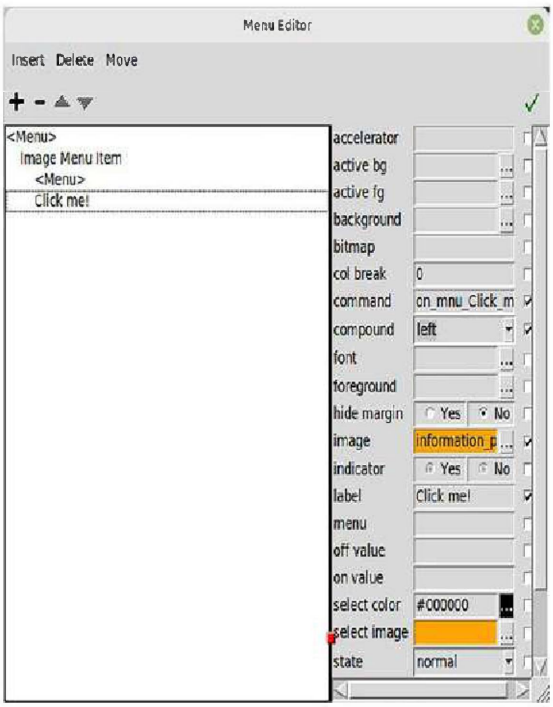


Figure 5.4: Menu Editor

Now, click on the image attribute button with three buttons. This will bring up a file dialog asking you to select the image:

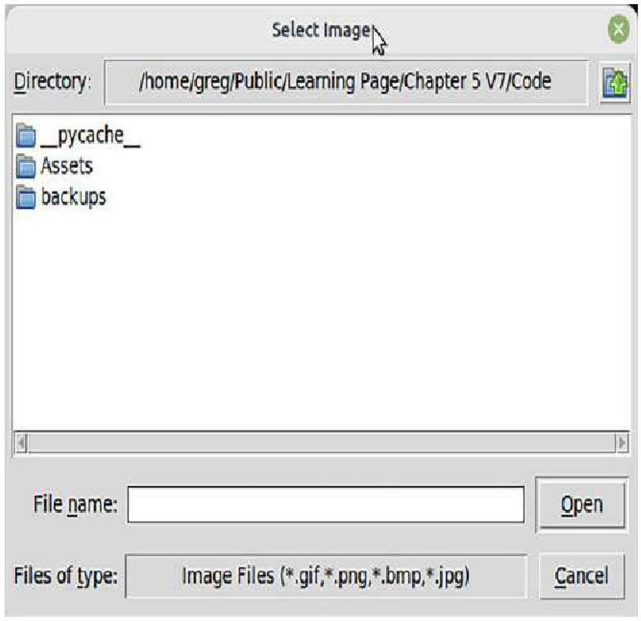


Figure 5.5: Select Image Dialog

Navigate to the Assets folder (where you have saved your images) and select the `information.png` image entry and click on the Open button. Finally, in the compound attribute, use the dropdown to select `left`. This will place the image to the left of the text. More on using the compound attribute in a few moments.

That is for our menu. Now, click on the green checkmark on the upper right of the menu editor to close the editor.

Next, we will add a few widgets that support Images.

When you use either a Button, Label, Tbutton, or TLabel, the process of embedding a graphic image is easy in PAGE. First, you would place and size your widget, set your text (if any) and (in the case of Buttons and TButtons) your callback as you normally do. The only extra thing that you need to do is select your image using the image attribute in the Attribute Editor, then select what type of placement you want between the text and image (the compound attribute). The process of selecting the image is almost exactly like we just did with the image in the menu.

We have created a small demo that visually shows the various compound settings and the effects on the placement between the image and the text. This works the same way on Buttons and Labels:

***Figure 5.6:** Compound Attribute demo*

As you can see, the position of the graphic image matches the value of the compound attribute. None causes the text to not be shown. You can also see that unless the graphic is really something special, you probably will not be using the centre compound attribute very often.

Now, we will start placing the other widgets. The first four widgets will go in a column going from near the top down.

First, we will place a standard Tk Button. Place it at x=170, y=30 and set the height to 48 and the width to 116. Make sure that the Alias is **`Button1`**. Set the compound attribute to the left. Click on the image attribute button with three dots and navigate to the **`Assets`** folder and select **`applications-graphics.png`** . Finally, set the text to **`Button`** if it isn't already.

Next, add a standard Tk Label. Place it at x=170 and y=110 and set the width to 116 and the height to 48 and set the Alias to **`Label1`**. Again, click on the image attribute button with the three dots and navigate to the Assets folder and this time select the file **`edit-clear.png`** . Now, make sure that the text attribute is set to **`Label`** and the compound attribute to the **right**. We will also set the relief to **groove**, so the label stands out a little bit.

We will add a ttk Tbutton next. Place it at x=170, y=166, height to 64 and width to 83 and make sure the Alias is **`TButton1`**. Set the compound attribute to the **top** and the

image to the `internet.jpg` file in the `Assets` folder. Make sure that the text attribute is set to `Tbutton`.

The last widget for this column will be a ttk TLabel. Place it at x=170, y=260, height to 62 and width to 82 and the Alias to `TLabel1`. Set the compound attribute to the **bottom** and the image to **system-run.png** in the **Assets** folder. Finally, make sure the text attribute to **Tlabel** and the relief to **groove**.

We will now add a column of Tk Labels to the left of the widgets.

Place the first Label at x=50, y=10, height to 91 and width to 109. Make sure the Alias is set to `Label2`. Set the anchor to "`e`", justify to '`right`', text to "`Tk Button - Compound left, Anchor w`" and the wrap length to **140**.

Place another Tk Label at x=50, y=110, height to 38, width to 108, and the Alias to `Label3`. Set the anchor to "`e`", justify to "`right`", the text to "`Tk Label - Compound right`" and the wrap length to **130**.

Again, place a Tk Label at x=50, y=177, height to 38, width to 120 and the Alias to `Label4`. Set the anchor to "`e`", the justify to "`right`", the text to "`ttk Button - Compound Top`" and the wrap length to **145**.

Finally for this group of Labels, place another at x=20, y=270, height to 38, width to 126, and the Alias to `Label5`. Set the anchor to "`e`", the justify to "`right`", the text to "`ttk TLabel - Compound Bottom`", and the wrap length to **145**.

Now, save your project.

Now, we will place another column of widgets just to the right of the first ones. For these, we will not change any of the attributes other than the location and size, because we will set all of them in code.

We will start with a standard Tk Button. Make sure that the `Alias` is `Button2` and set the position to x=300, y=30 and the height to 48 and the width to 116.

Now, place a Tk Label and set the position to x=300, y=110, height to 42 and width to 102. Also, set the relief to `groove`. Make sure the Alias is `Label6`.

Next, place a ttk Tbutton and set the position to x=300, y=110, height to 42, and width to 102. Set the Alias to `TButton2`.

Now, place a ttk TLabel at x=300, y=260, height to 62, and width to 82. Set the relief to `groove` and set the Alias to `TLabel2`.

Again, save your project and generate the GUI and Support Python files.

We need to create another column of Tk Label widgets to the right of the last column of widgets. This time, we will just need to place and size the Labels in the designer and the rest will be done in code.

Place the first Label at x=440, y=7, height to 91, and width to 109. Set the Alias to `Label 7`.

Place the next Label at x=440, y=86, height to 91, and width to 109. Set the Alias to `Label 8`.

Place the next Label at x=440. Y=156, height to 91, and width to 109. Set the Alias to `Label 9`.

Place the last Label at x=440, y=245, height to 91, and width to 109. Set the Alias to `Label 10`.

We are almost finished with the design of our project. Now, we are going to place a Tk Checkbutton and a Tk Radiobutton. We will use custom graphics for both the widgets. We will set the graphics for both in the designer.

Place a Tk Checkbutton at x=100, y=360, width to 127 and height to 33. Now, make sure the Alias is **Checkbutton1**. Change the text to "**Check**" and make sure the variable is **che46**. Ensure the compound attribute is set to "**left**" and find the indicator attribute and click on it from **Yes** to **No**. (This will make sure that our graphics are used and the standard box is not seen.). Set the border width to **0**, so the relief is not raised. Now, for the image attribute, click on the three dot button and navigate to "`checkbox-unchecked.png`" in the Assets folder. It is recommended to have a little bit of extra space between the graphic and the text, so set the x pad to a value of **4**. Finally, find the select image attribute, click on that three-dot button and select "`checkbox-checked.png`" in the Assets folder.

In the same manner, add a Tk Radiobutton at x=100, y=400, width to 138, and height to 33. Make sure the Alias is set to `Radiobutton1`, the text to "`Radio`", the variable to "`selectedButton`" and the value is set to 1. Again, set the border width to 0 and the indicator attribute to `No`. Set the x pad attribute to **4**, again to give a little bit of space between the image and the text. Like we did with the Check button, set the image attribute to "`radio-unchecked.png`" in the Assets folder and then select the image to "`radio-checked.png`" in the Assets folder.

Save your project and generate your Python files, so we can move on to the code.

Writing the Code

We will need to add a few items to the imports section. Those that PAGE provides will be shown in normal font, but those that you need to add will be in bold:

```
import sys

import os

import platform

import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *

from tkinter import messagebox


import graphics
```

Notice that we have left a blank line between the import **messagebox** line and the import graphics. We do that just to make sure that we do not accidentally delete that line, since the program will not work without it. You can leave the blank line out if you want.

Now, as we have done before, need to modify the main function. All we have to do is add one line just before the end of the function:

```
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 = graphics.Toplevel1(_top1)
    startup()
```

```
    root.mainloop()
```

We now should include the startup function. We start the function with setting a global variable named version, and then print the Python version and the version of the program to the terminal. Next, we use a variable from the GUI module named _location. This is the full path to the source code that PAGE provides. This will provide part of the path to our Assets folder. We will use the join method from the **os.path** library to create an operating system-safe full path to our graphics. This way, our program always knows where our graphics are, even if later on we move the program folder:

```
def startup():

    global version

    pv = platform.python_version()

    print(f"Running under Python {pv}")

    # Set the path for the icon files

    version = '0.0.2'

    print(f"Version: {version}")

    progloc = graphics._location

    global graphloc

    graphloc = os.path.join(progloc, "Assets")

    print(f'Program Location: {progloc}')

    print(f'Graphics Location: {graphloc}')
```

Next, we define four global variables and call them img1, img2, and so on. We need to provide these global variables since Python has a garbage collection system built in and runs while the program runs, clearing variables that are not being used. If we didn't, the garbage collection process will "forget" about the images we want on our widgets:

```
    # ====================================

    # Now manually add the graphics to the remaining widgets

    # ====================================

    # Define the global variables we need to keep
```

```
# ===================================

global img1, img2, img3, img4
```

Now, we can start to configure the four widgets that we just placed into the form. These are all the steps we go through when we assign the graphics in the designer, but we are doing it in code. First, for the tk Button widget Button2, we set up the filename, by adding the name of the graphic file to our global graphics path using the os.path.join function again. Then, we assign the filename to the global variable using the **tk.PhotoImage** function. PAGE does this for us when we embed the images within the designer. Since we are doing this ourselves, we have to do it manually. Then, we set the image attribute and the compound attribute:

```
filename = os.path.join(graphloc, 'edit-paste.png')

img1 = tk.PhotoImage(file=filename)

_w1.Button2.configure(image=img1)

_w1.Button2.configure(compound=RIGHT)

_w1.Label7.configure(anchor=W)

_w1.Label7.configure(text='Tk Button - Compound Right')

_w1.Label7.configure(wraplength=145)
```

Now, we go through the same process with the rest of the four widgets. We left a blank line between each block of code so it is easier to keep track of where we have been when we are adding the code. Feel free to leave the blank lines in or not:

```
filename = os.path.join(graphloc, 'edit-undo.png')

img2 = tk.PhotoImage(file=filename)

_w1.Label6.configure(image=img2)

_w1.Label6.configure(compound=LEFT)

_w1.Label8.configure(anchor=W)

_w1.Label8.configure(text='Tk Label - Compound Left')

_w1.Label8.configure(wraplength=12


filename = os.path.join(graphloc, 'insert-text.png')

img3 = tk.PhotoImage(file=filename)
```

```
    _w1.TButton2.configure(image=img3)
    _w1.TButton2.configure(compound=BOTTOM)
    _w1.Label9.configure(anchor=W)
    _w1.Label9.configure(text='ttk TButton - Compound Bottom')
    _w1.Label9.configure(wraplength=120)

    filename = os.path.join(graphloc, 'notebook.png')
    img4 = tk.PhotoImage(file=filename)
    _w1.TLabel2.configure(image=img4)
    _w1.TLabel2.configure(compound=TOP)
    _w1.TLabel2.configure(anchor=CENTER)
    _w1.Label10.configure(anchor=W)
    _w1.Label10.configure(text='ttk TLabel - Compound Top')
    _w1.Label10.configure(wraplength=145)
```

When we designed the menu bar, we defined a callback for when the user clicks on the menu item (remember, there was only one item). Rather than just have our program print that the user clicked on the menu, we will use the messagebox built-in feature of Tk to provide a message box to the user. (we will talk about other options for the message boxes in an upcoming chapter:

```
def on_mnu_Click_me(*args):
    print('graphics_support.on_mnu_Click_me')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Graphics Demo"
    msg = "You clicked on the menu item"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

Save your code. You will see the project as shown in *Figure 5.1*.

Conclusion

In this chapter, you learned how to include graphic images in Button widgets, Label widgets, Checkbuttons, and Radiobuttons. You also learned how to do it either in the PAGE designer or in code.

In our next chapter, we will expand our knowledge about menus and talk about context menus or popup menus.

Take a break and then come back.

Multiple Choice Questions

1. Can you place your graphics anywhere?
   a. Yes
   b. No
   c. Sort of
2. What does the compound attribute do?
   a. Allows you to put more than one graphic image on a widget.
   b. Allows you to control where the graphic image is placed in the widget in relation to the text.
3. PAGE provides a variable that holds the path to the location of the program when run. What is the name of this variable?
   a. path
   b. location
   c. _location
   d. There is no such thing.

Answers

1. C – Sort of. You can actually keep your graphics anywhere, but it is good programming practice to place it either within the project folder directly or in a sub-folder within the project folder. That way, it is easy for Python to find the images and if you move the project folder, the path to the images are always the same.
2. B – It allows you to control where the image shows up in relation to the text.

3. C – The name of the variable is _location. You need to include the name of the main project file with it. For our graphics program, it was graphics._location.

Questions

1. Why do you need to create global variables when using graphics in code?
2. Does the Toplevel form support an background image?

Answers to the questions

1. We have to create global variables to hold the graphic image created from the tk.PhotoImage function so that Python's garbage collection doesn't delete that and keeps it from being seen.
2. No, the Toplevel has no image attribute.

Keywords

- Graphics
- images
- tk.PhotoImage
- embedding
- path

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

Menus and Popup Menus

Introduction

We have learnt a great deal so far, and now we are going to learn about Menus, Popup menus and Button menu bars. Many programmers today write GUI programs with only Button bars as menus, assuming that the textual menus are old school. However, they also use popups or contextual menus to extend the capabilities of the program or the widgets themselves.

What you will learn

In this chapter, we will learn the following topics:

- Create a form level menu
- Create a Popup menu
- Using Tooltips

- Binding keystrokes and Mouse events
- Possible Menu Items
- Radiobuttons
- Checkbuttons

- Commands and Cascades
- Bindings
- Accelerators
- Creating the GUI
- Writing the Code

Menu overview

Take a look at PAGE. On the main window is the 'old style' menu. Yes, it could have been created with a Button bar style menu, but when you look at the sheer number of options that the menu holds, it really does not lend itself to a Button bar.

Take a look at a demo we will be creating to show the Menu and Popup Menu abilities that PAGE and Tkinter offer:

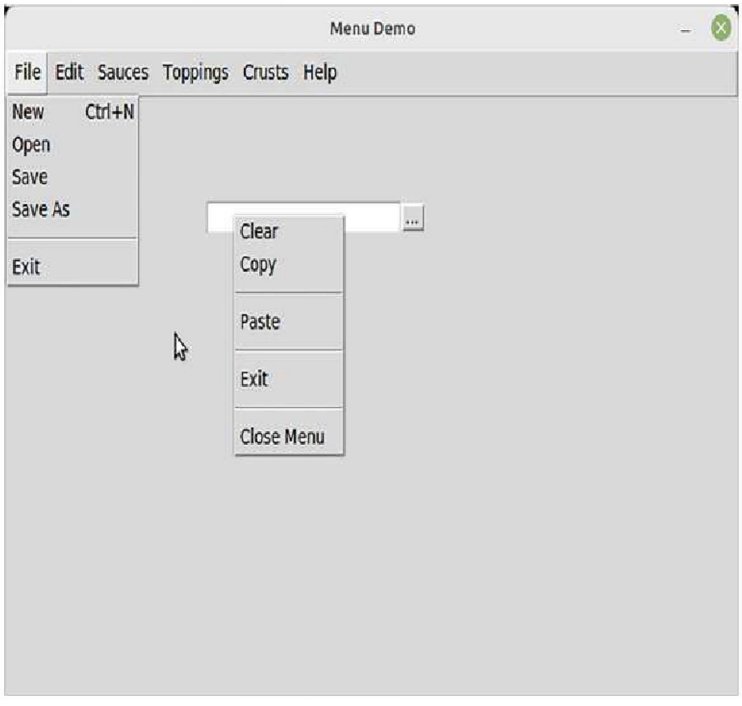


Figure 6.1: Menu Demo

You will see that the form-level and popup menus look remarkably the same. The editor is the same for both. In short, to understand how to create and properly use popup menus, you have to understand form level menus.

Before we get started

We discussed some of these items in earlier chapters, but we'll go through them and add them to our knowledge. In the fictional demo for this portion of the chapter, we mixed a standard `File` | `Edit` | `Help` style menu with one that would work for a Pizza shop application, where the various menu items could represent the crust styles, sauce type, toppings, and so on.

Possible menu items

Our menus can contain various types of entries. We will look at them here. We will start with the form level menu from the demo we just showed:

Figure 6.2: Menu Demo Snippet

Look at the top of the image and see the menu items. Each will drop down a series of menu items. The items at the top are all called cascades. This means that they contain one or more items. This can also be embedded within another cascade.

In the previous image, you will see the commands that are children of the cascade named 'File'. Just before the last item in this dropdown portion of the cascade, you will see a 3D line. This is known as a **separator**. These commands all should have a command attribute, which is the callback that will be called when that item is

clicked. Just like the command attribute for a Button widget.

Radiobuttons

Our menu cascades can use Radiobutton items, just like command items. It really doesn't make any sense to create a Radiobutton item by itself. You should have more than one Radiobutton item. For the sauces option in our fictional menu, there would only be one sauce on any given pizza, just as you would only have one crust type

per pizza. Remember that Radiobuttons represent a one-of-many option set. Only one may be selected at a time.

The Radiobutton menu items are programmed pretty much like a regular Radiobutton widget that we would put on the form:

Figure 6.3: Radiobutton Items

Checkbuttons

Like the Radiobutton items, a cascade can also have Checkbutton children:

Figure 6.4: Checkbutton Items

In our fictional menu, the Checkbutton children give immediate attention to which Topping options the customer wants. The Checkbutton items are a many-of-many representation of a number of options. Unlike Radiobuttons, any or all may be selected at a time.

Cascade items can have Commands and Cascades in any order as shown in *Figure 6.5*:

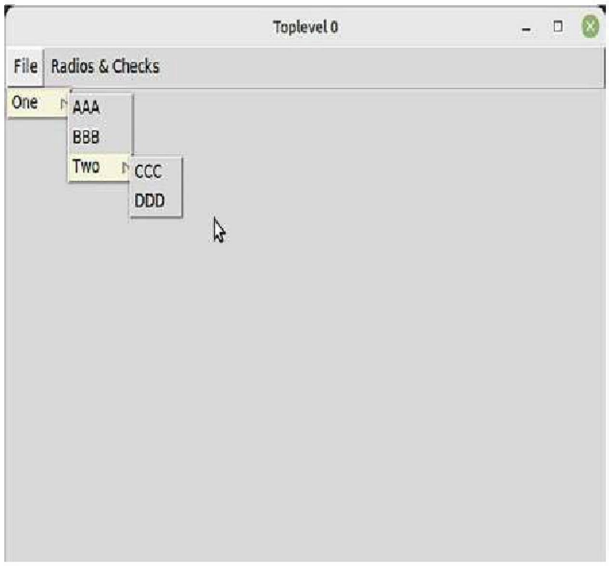


Figure 6.5: Cascades with Cascades

You can also place Radiobuttons and Checkbuttons under a cascade:

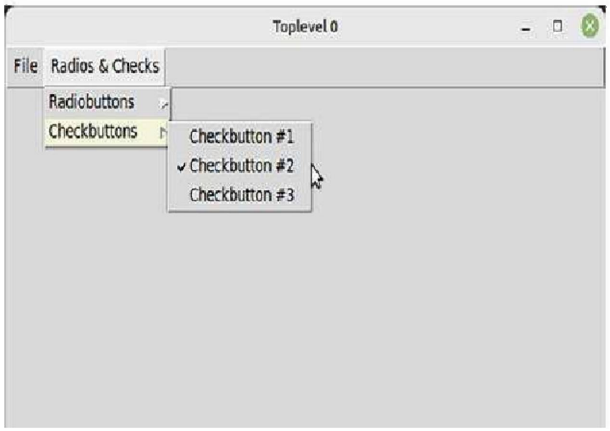


Figure 6.6: Radiobuttons and Checkbuttons

If you end up with so many options in a menu cascade, do not worry, you can break the items into columns. To use this type of menu, set the '`col break`' (column break) attribute to 1 for the first option that you want to start the new column. In the case of the following image, it would be `Option 4`:

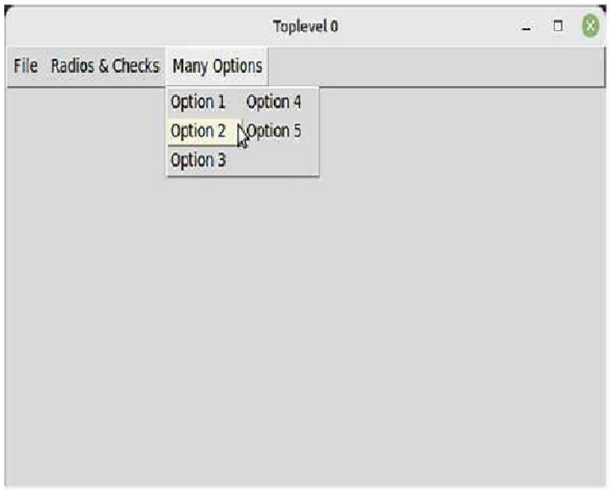


Figure 6.7: Items in multiple Columns

Popup menus can have all the same abilities.

Bindings

When we use the command attribute of a Button or a Checkbutton, we are effectively creating a binding that takes the mouse event and binds it to the particular widget. The Binding process also adds the ability to keep track of various mouse button events, mouse movements, and keystrokes to callback functions. When we create popup menus, we usually bind them to the third mouse button (right mouse button for most people) which will cause the popup menu to ... er ... popup.

Accelerators

Accelerators in a menu or pop-up menu allow a special keystroke or combination of keyboard presses to be mapped to a particular menu function. Many times, the F1 key of the keyboard will be mapped to the Help function of a particular program. The key combinations of *<Ctrl> + N*, *<Ctrl> + O* and *<Ctrl> + S* will be mapped or bound to File New, File Open, and File Save for many programs to make things easier for the user to do normal tasks. However, when we create an accelerator for various menu items, it is just a text representation of the keystrokes to aid the user. We have to take additional steps to create the bindings for these. However, PAGE has a bug in version 7.4 and 7.5 that prevents the use of accelerators, so we will not try to use them in this chapter.

We will go through all these things and more as we create our GUI program. However, before we create our GUI, we need to know what items our menus and pop-up menus will have and their callbacks. We have created a text-based outline of our

menu and popup menu. For the form level menu, the items that are un-indented are cascades and those that are indented are command items. The items in Parentheses (if any) are accelerators and the last thing on the line is the command callback:

```
File
    New - on_mnu_FileNew
    Open - on_mnu_FileOpen
    Save - on_mnu_FileSave
    Save As - on_mnu_FileSaveAs
    ------------------
    Quit - on_mnu_FileQuit
Help
    About - on_mnu_HelpAbout
    -----------------
    Help - on_mnu_HelpHelp
```

Our popup menu will be much simpler since this time we will not use cascades, just command items, along with the command callback information:

```
Copy - lambda : on_pop_Copy(args[0])
Cut - lambda : on_pop_Cut(args[0])
Paste - lambda : on_pop_Paste(args[0])
Clear - lambda : on_pop_Clear(args[0])
Quit - lambda : on_pop_Quit(args[0])
-----------------
Close Menu - lambda : on_pop_CloseMenu(args[0])
```

Now that we have designed our menu and popup menu items, we can start our GUI design.

Creating the GUI

To create the project file for this chapter, start PAGE and name the project "`menudemo.tcl`". Drag your Toplevel form to somewhere near the center of your screen.

Next, add a standard Tk Entry widget to your form. It really doesn't matter where you place it, but I put mine at x=120 and y= 120, the width to 166 and the height to 23. Finally, set the `textvar` to "`EntryData`". Leave the Alias as "`Entry1`".

Now, click on a blank spot on the Toplevel form so the Attribute Editor reflects the Toplevel form. Then, click on '`<click to edit>`' for the menu editor:

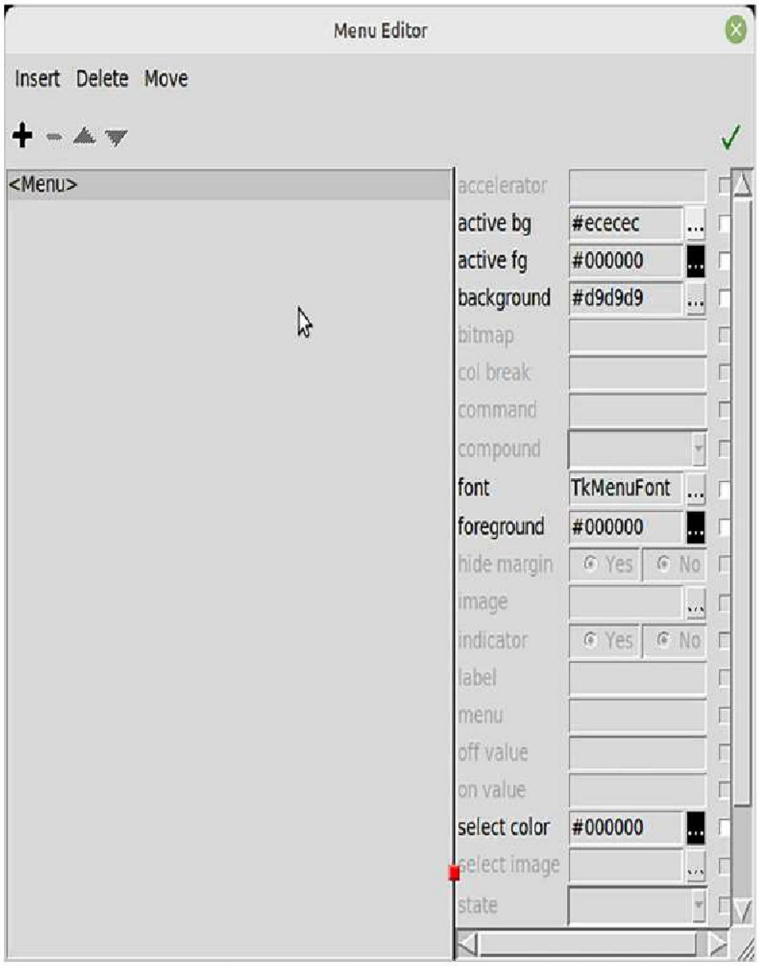


Figure 6.8: Empty Menu Editor

Create the first Cascade item (File), then add all six command items as follows. Again, the line of dashes represents a separator. Use that Accelerator attribute to set the accelerator text (in Parentheses) and the command attribute for the command items. Be sure that when you add a new item the parent or sibling is selected in the editor:

```
File
    New - on_mnu_FileNew
```

```
Open - on_mnu_FileOpen

Save  - on_mnu_FileSave

Save As - on_mnu_FileSaveAs

------------------

Quit - on_mnu_FileQuit
```

When you are done with this, the menu editor should look something like this:

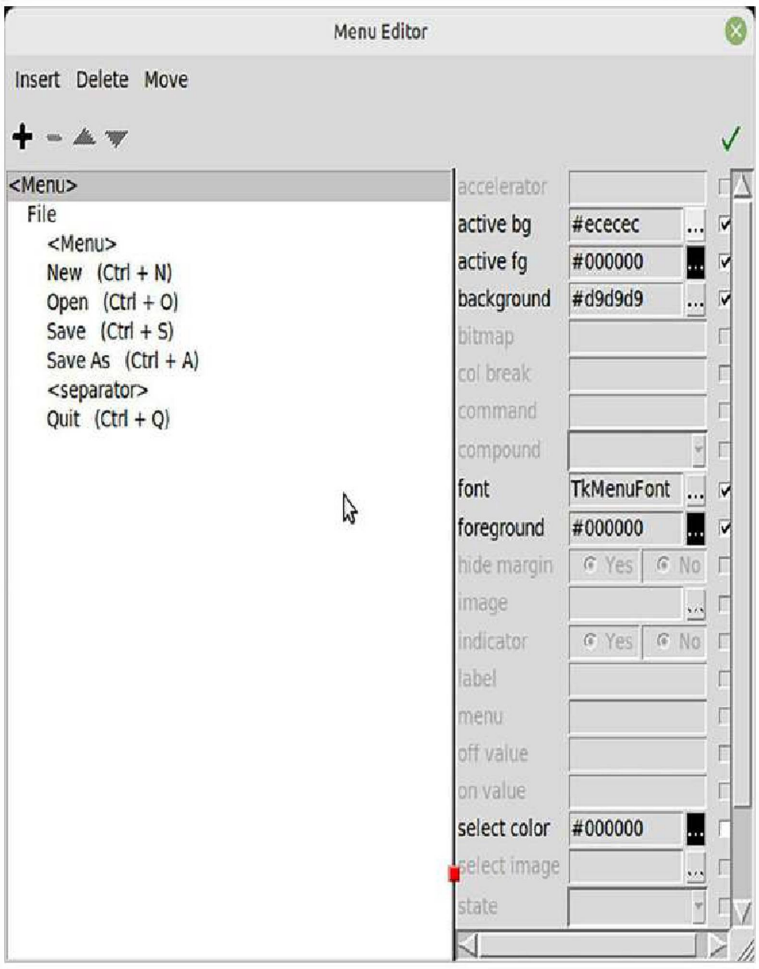


Figure 6.9: Menu Editor Part 1

Now, add another cascade item (be sure to click on the very top <Menu> first) and add the Help portion of the menu.

After you have added the help items, your menu editor should look something like this

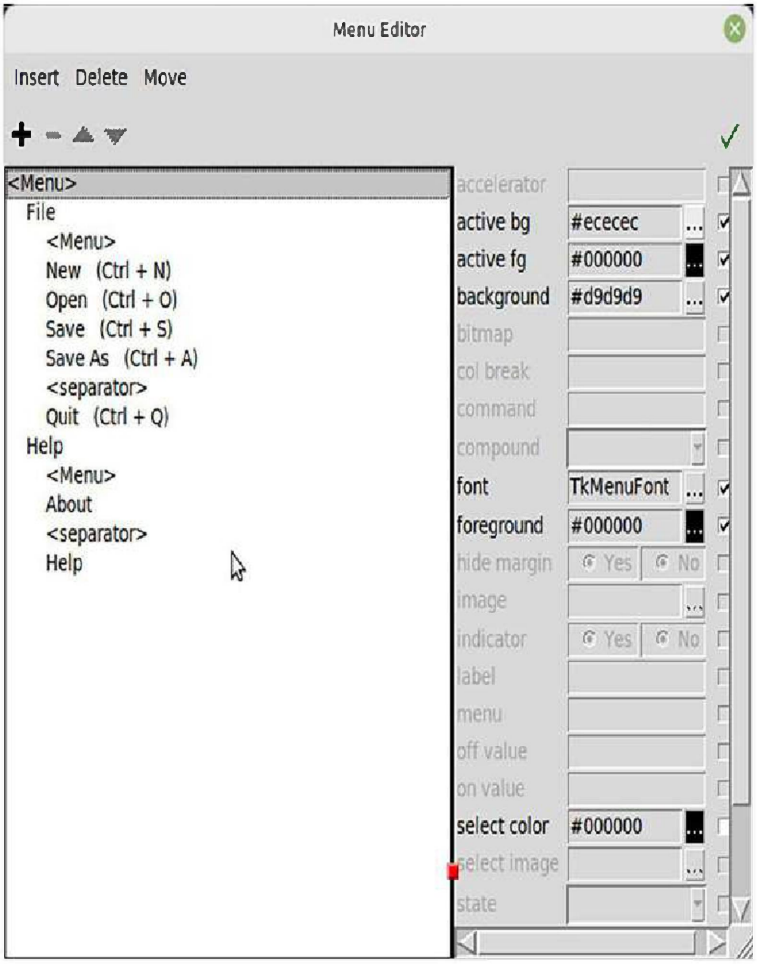


Figure 6.10: Completed Menu

Next, we will create the popup menu. You can find it in the Widget Toolbar under the Tk Widgets section named Popupmenu. Click on that.

> *Note: If you are using PAGE 7 to PAGE 7.5, simply clicking on the Popupmenu widget in the Toolbar is enough to create it in the project. Under PAGE 7.6, the Popupmenu is not simply tied to the project, but to a specific Toplevel form. You will need to click on the Popupmenu widget in the toolbar first, then on the Toplevel form you wish to tie it to.*

If you look in the Widget tree, you will see the entry for the **Popupmenu**. There is no indication on the Toplevel form that there is a **Popupmenu** widget:

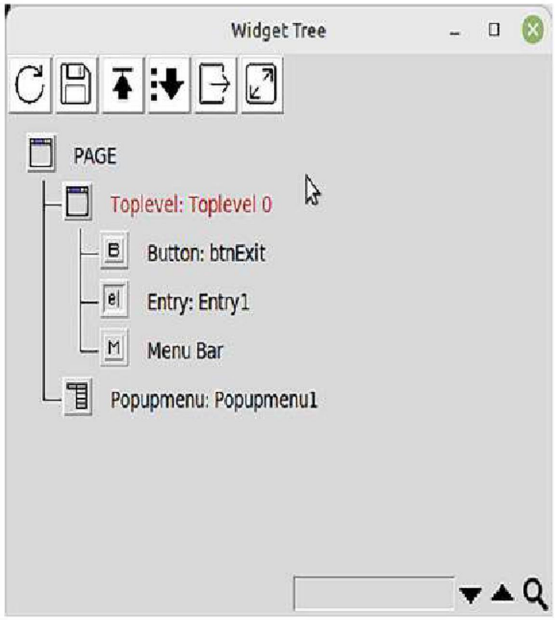


Figure 6.11: Popup menu in Widget Tree

To bring up the menu editor for the Popup menu, simply double click on the Popup menu widget in the Widget Tree. The menu editor will be blank. Now, add the following items, all as command items (no cascade items) as follows:

```
Copy - lambda : on_pop_Copy(args[0])

Cut - lambda : on_pop_Cut(args[0])

Paste - lambda : on_pop_Paste(args[0])

Clear - lambda : on_pop_Clear(args[0])

Quit - lambda : on_pop_Quit(args[0])

-----------------

Close Menu - lambda : on_pop_CloseMenu(args[0])
```

You might wonder what the "lambda :" and the "`(args[0])`" parts are all about.

Tkinter normally expects callbacks do not have parameters. However, there is a way around it, which is to use the lambda keyword as part of the callback definition. We will be passing a value into the callback from each of the popup items, which will eventually be a 1 for the Entry widget #1 and a 2 for the second Entry widget which we will be adding in a little bit. By doing this, we can use the same popup menu for multiple widgets on our form.

After you have done that, your menu editor should look something like this:

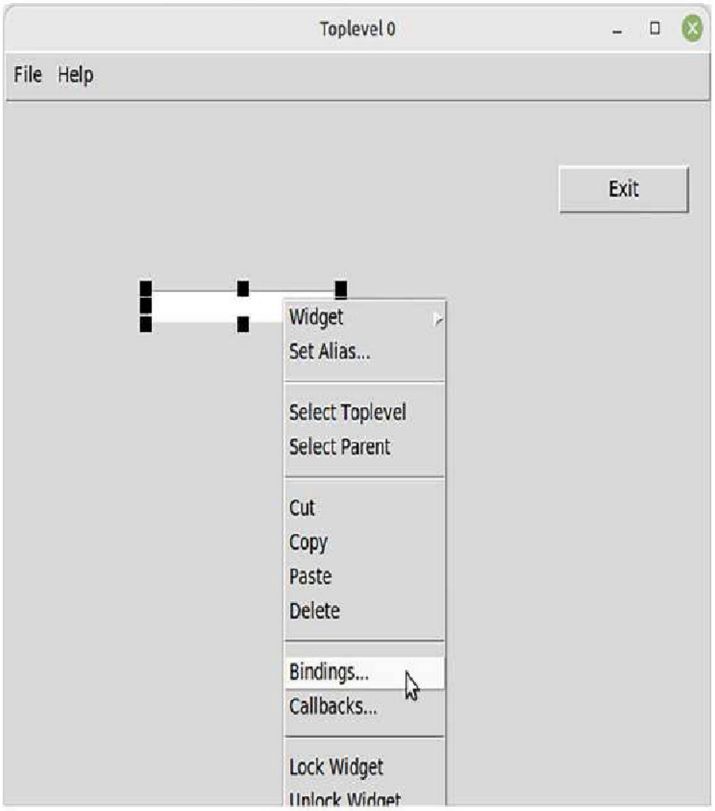


***Figure 6.12:** Completed Popup menu*

Remember that you won't see any indication of the popup menu in the Toplevel form. It will not show up until we create the binding for it with the Entry Widget.

There are two ways to create bindings for widgets. One in the code which we'll see in a few moments and one using the PAGE Bindings editor. We will use it that way now.

Going back to our Toplevel form, select the Entry widget and use the Mouse Button 3 (right click for most people) to click the widget again. This will bring up the PAGE widget context popup menu:

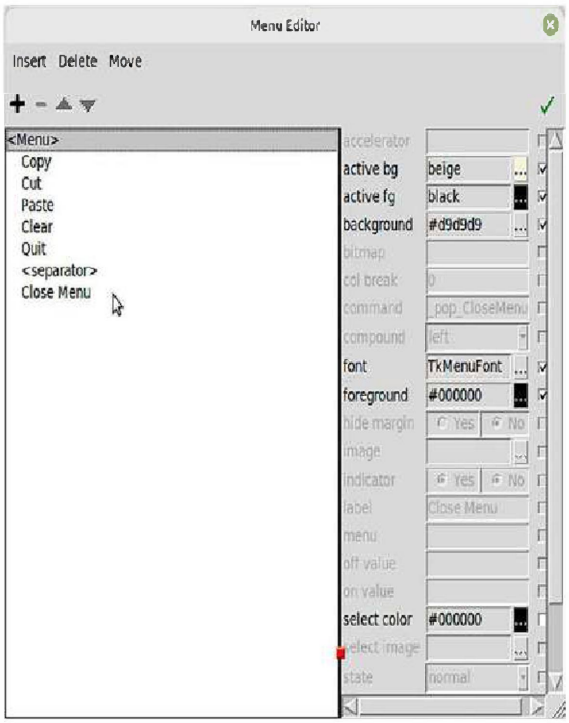


Figure 6.13: Widget Context Menu

Find the Bindings option and click on it. This will bring up a new editor called the Bindings editor:

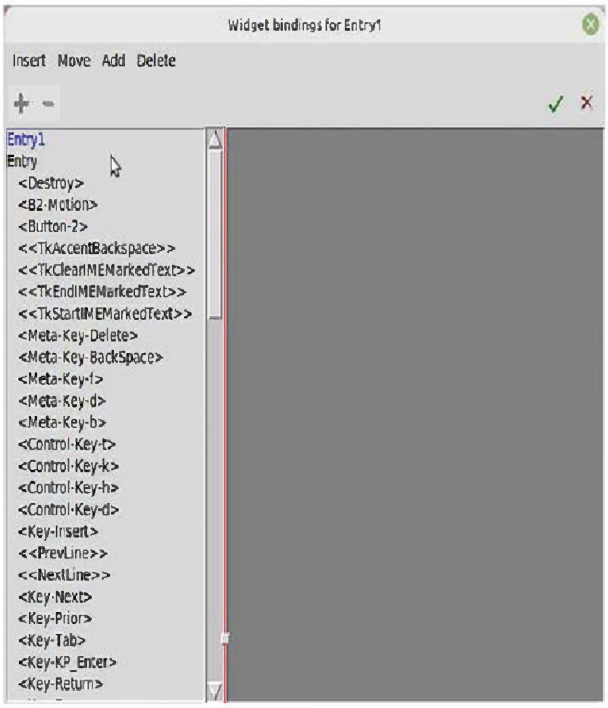


Figure 6.14: Bindings Editor

Once we see the Bindings editor, we need to add the event we want to use to trigger the action, which in this case will be to call the popup menu.

Near the upper left corner of the editor, you will see a large "+" sign. This will show another context menu that gives you several possible event items:

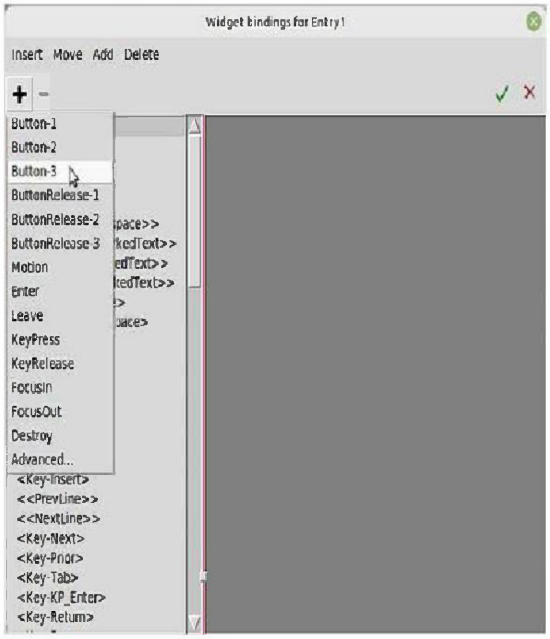


Figure 6.15: Binding to an Event

Select the **`Button-3`** event, and your editor will change it to something like this:

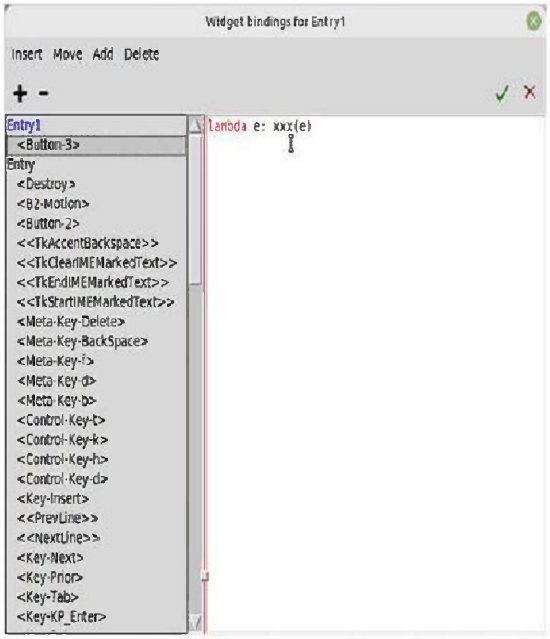


Figure 6.16: Callback Skeleton

This will create a skeleton containing the command that will be used to create the callback function for your binding. This is similar to the command attribute that we have used many times before. However, this last step is important. To enter the name of the callback, you must click just after the "xxx" in the right pane to place the cursor to be able to type in the editor. Once your cursor is in place, use the backspace key to delete the three "*x*" characters and then type your new callback function name. In this case, we will be calling it "**`popup1(e, 1)`**". Since the "(e)" portion is already there, just enter "**`popup1`**", then move the cursor behind the "e" but before the ")" and add ", 1" as shown in *Figure 6.17*:

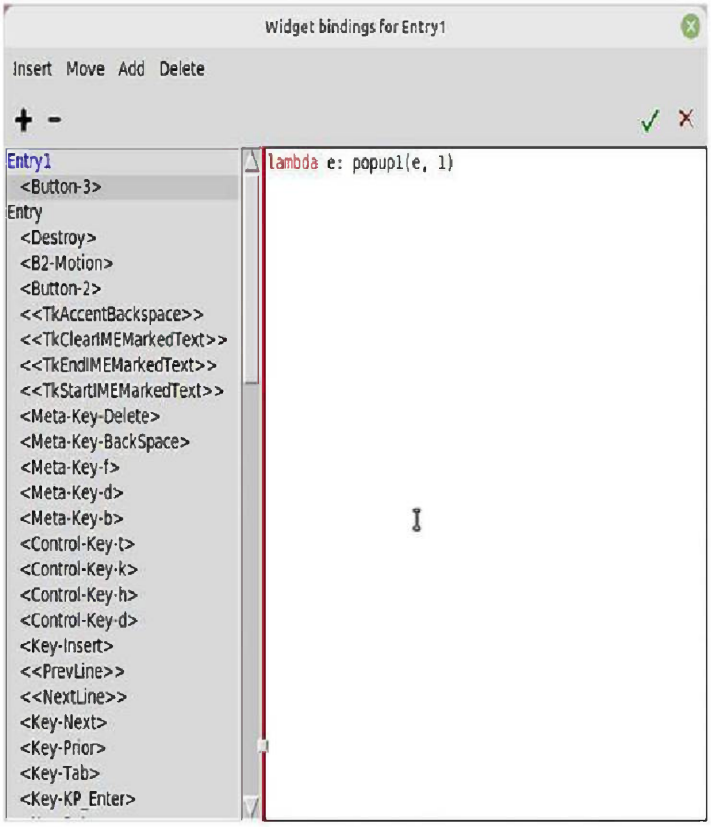


***Figure 6.17:** Callback complete*

Just in case you are interested, the "e" stands for the event, which will send in information about the event that triggered the callback.

Now, click on the green check mark to save your callback information and dismiss the Binding editor.

Again, add another Tk Entry widget to the form, place it at x=120 and y=210, width = 166 and height = 23, and leave the Alias to Entry2 but set the text var to "**`EntryData2`**".

Finally, we will need to create a graphical menu bar using Buttons and images:

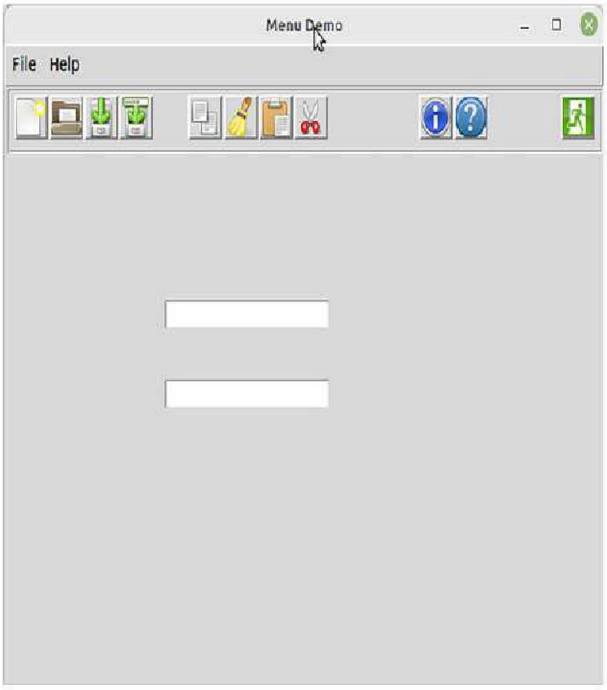


Figure 6.18: Project completed

The menu bar is fairly simple to do. It is just a frame that acts as a container to hold several standard Tk buttons, each having an image that depicts what the button is going to do. For this project, we will add tooltips, text that appears over the widget when the mouse cursor hovers over the widget. When dealing with menu buttons especially, it is a good idea to include a tooltip.

We will start with the frame.

Place a standard Tk frame widget on the Toplevel form and set the location to x=4, y=2, and set the width to 595 and the height to 48. Set the relief to **groove**.

We will be placing 11 standard Tk Buttons from left to right in the frame. All the buttons will have a width of 36 and a height of 36. All the graphic images are 32x32 pixels and are included in the Assets folder.

The first will be Aliased as **btnFileNew**. Set the x position to 5 and its y position to 3. Set the width to 36 and the height to 36. Set the command to **on_btnFileNew**, the image property to **document-new.png** (in the Assets folder), and the tooltip text to **File New**.

Add another Tk Button and set the Alias to **btnFileOpen**. Set the x position to 40 and the y position to 3. (All the following Buttons will be set to a Y position of 3.) Again, set both the width and height to 36. Set the command to **on_btnFileOpen**, the image to **document-open.png**, and the tooltip to **File Open**.

Now, add another Tk Button and set the Alias to **btnFileSave**, the x position to 75, the y position to 3, and the width and height to 36. Set the command to **on_btnFileSave**, the image to document-save.png, and the tooltip to File Save.

Add another Button and set it as Alias to **btnFileSaveAs**, the x position to 110 and the y position to 3. Do not forget to set the width and height to 36. Set the command to **on_btn_FileSaveAs**, it is an image to **document-saveas.png** and the tooltip to **File Save As**.

That takes care of the File section of the menu bar. Now, we will create four more buttons for the Edit section. Remember that the Y position for all the buttons is 3 and the width and height are all 36. We won't include them in the next steps.

Add another Button and set the Alias to **btnCopy**, the X position to 179, the command to **on_btnCopy**, the image to **edit-copy.png**, and the tooltip to **Copy**.

Add another Button and set the Alias to **btnClear**, the x position to 215, the command to **on_btnClear**, the image to **edit-clear.png**, and the tooltip to **Clear**.

Again, add another Button and set the Alias to **btnPaste**, the x position to 250, the command to **on_btnPaste**, the image to **edit-paste.png**, and the tooltip to **Paste**.

Add another Button and set the Alias to **btnCut**, the x position to 285, the command to **on_btnCut**, the image to **edit-cut.png**, and the tooltip to **Cut**.

We will deal with the Help section next.

Place another Button and set it as Alias to **btnAbout**, its x position to 410, the command to **on_btnInfo**, the image to **information.png**, and the tooltip to **About**.

Place the next to the last Button and set Alias to **btnHelp**, the x position to 445, the command to **on_btnHelp**, the image to **help-contents32.png**, and the tooltip to **Help**.

Finally, place the last button into the frame and set it is Alias to **btnAppExit**, its x position to 552, the command to **on_btnExit**, the image to **system-log-out.png**, and the tooltip to **Exit App**.

Save your project and generate the Python files. You can close PAGE at this point and we will get started with our code.

Writing the code

Many of the functions in our code will be essentially the same as other functions, so you can copy and paste and only change the things that are needed for that particular function.

As always, we need to start with the import section. As we have done in previous chapters, we will now show the lines you need to add in bold and those that PAGE generates for us in the normal face:

```
import sys

import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *

from tkinter import messagebox
```

We need to import the **messagebox** module from tkinter to provide popup messages as notifications to the user. We will not be trying to do any file manipulation in this project, we will just respond with the **messagebox**.

As we have done in the previous chapter projects, we need to create a call to a setup function from the main function to do the start-up processes:

```
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 = menudemo.Toplevel1(_top1)
    setup()
    root.mainloop()
```

The startup function does not get created by PAGE; since it is unknown what, if any, things need to be done. You will need to provide the entire function on your own. In this case, we will be doing is creating the event binding for the Entry2 widget that ties the mouse-3 click to the popup menu. It is pretty much the same thing as using the bind menu within PAGE itself:

```
def setup():

    # binding for Entry2 widget

    _w1.Entry2.bind('<Button-3>', lambda e: menudemo.popup1(e, 2))
```

The code for the `Exit` button callback is easy this time as it was in the previous chapters. It is just a single line calling `sys.exit` to end the application:

```
def on_btnExit(*args):

    print('menudemo_support.on_btnExit')

    for arg in args:

        print('    another arg:', arg)

    sys.stdout.flush()

    sys.exit()
```

When PAGE creates the skeleton functions for us, the order is mostly based on the order that the widgets were added. In this case, the main form menu callbacks will usually show up first. When we created this version of the demo, the File Menu was first:

```
def on_mnu_FileNew(*args):

    print('menudemo_support.on_mnu_FileNew')

    for arg in args:

        print('    another arg:', arg)

    sys.stdout.flush()

    titl = "File New"

    msg = "You asked to create a new empty file."

    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

When we use the message feature of Tkinter, we should provide a title for the `messagebox`, a message to let the user know something, and optionally, which icon

to use which usually should match the type of message box it is and the parent Toplevel form so the **messagebox** will be centered on that form. In our case here, we only have one form, so we will use **_top1**.

There are three types of **messagebox** dialog boxes available. An Information style, an Error style, and a Warning style. To call the Information style, we use **messagebox.showinfo()**. The Error **messagebox** is called by using **messagebox.showerror()** and the warning **messagebox** is called by using **messagebox.showwarning()**. The information icon is added by using **messagebox.INFO**, and the error icon is **messagebox.ERROR**, and the warning icon is **messagebox.WARNING**.

We set the string that we want as a title to a variable, then we set the message to another variable so that we do not have to enter them directly in the call. You can, however, enter them directly in the call, but it's easier to do it using variables. It also keeps your code neater.

I said earlier that many of the functions that we will be using are the same, so you can simply copy the last three lines of the menu function and paste it into the next one, then change the first two lines to match the need for the new function, which would be the line defining the title and the line defining the message. For these functions, the call to the **showinfo messagebox** will be the same:

```
def on_mnu_FileOpen(*args):
    print('menudemo_support.on_mnu_FileOpen')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "File Open"
    msg = "You asked Open an existing file."
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The **FileQuit** callback is just like our button exit callback:

```
def on_mnu_FileQuit(*args):
    print('menudemo_support.on_mnu_FileQuit')
    for arg in args:
        print('    another arg:', arg)
```

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

Here is the callback code for the `FileSave` and `FileSaveAs` functions:

```
def on_mnu_FileSave(*args):
    print('menudemo_support.on_mnu_FileSave')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "File Save"
    msg = "You asked to Save a file"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)

def on_mnu_FileSaveAs(*args):
    print('menudemo_support.on_mnu_FileSaveAs')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "File Save As"
    msg = "You asked to save a file with a new name."
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The next two callbacks are for the help functions:

```
def on_mnu_HelpAbout(*args):
    print('menudemo_support.on_mnu_HelpAbout')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Menu Help"
```

```
    msg = "You asked for Application Information from Help|About"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)

def on_mnu_HelpHelp(*args):
    print('menudemo_support.on_mnu_HelpHelp')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Menu Help"
    msg = "You asked for Help from Help|About"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

That is it for our main menu code. It would be a good idea to save your code module before moving on.

Now, we are going to deal with the popup functions. It is important to remember that we are dealing with two possible widgets that can call the same function. When the function is being triggered from the Entry1 widget, there will be a 1 sent in as a parameter and if triggered from the Entry2 widget, there will be a 2 sent in.

The way this works is when we used the **lambda** function. Remember that I said earlier that Tkinter does not expect any parameters to be passed. If you try to do it without the lambda, the callback will be called immediately and will throw an error.

When the event is triggered, it calls the function **popup1** located in the GUI file. That is the function that not only creates the popup on demand but also causes the popup menu to be displayed. When you respond with a mouse click, the proper callback is then triggered.

When the callback is entered, the parameter is sent into the callback in the ***args** parameter. This allows you to pass in any number of parameters in any order. In our case, there is only one parameter. However, **args** is a **tuple**. So, to extract any (or all) of the parameters we must access it with an index. Since we know that there will only be one parameter sent in, we can get it by using **args[0]**.

The first thing we do is get the parameter and assign it to the variable **which**. Then, we use a simple **if** condition tree testing the variable to see if it is equal to 1 or 2, then dealing with whatever the value is and doing the proper steps for that value:

```
def on_pop_Clear(*args):
    print('menudemo_support.on_pop_Clear')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    # Clear the data in the Entry widget
    which = args[0]
    if which == 1:
        _w1.EntryData.set('')
    else:
        _w1.EntryData2.set('')
```

So, the Clear function will simply set the Entry text to an empty string (''). We use the which value to set the proper widget.

The **CloseMenu** callback simply takes the parameter and since the callback was called, the menu closes. If we did not have this option, the menu will continue to be displayed until one of the options is selected. If we did not have this, there would be no way to remove the menu if the user chose not to pick any of the menu options:

```
def on_pop_CloseMenu(*args):
    print('menudemo_support.on_pop_CloseMenu')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    # We don't have to do anything in this callback.  Just calling the
    # callback will close the popup menu
```

The **Copy** function will take anything in the Entry Widgets' text var and place it into the Tkinter system clipboard. We need to clear the clipboard first, then call **clipboard_append** to put it into the clipboard. We use the variable to determine which of the **textvars** to get the data:

```
def on_pop_Copy(*args):
```

```
    print('menudemo_support.on_pop_Copy')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    # Get the data from the Entry widget and place it on the clipboard
    which = args[0]
    if which == 1:
        data = _w1.EntryData.get()
    else:
        data = _w1.EntryData2.get()
    root.clipboard_clear()
    root.clipboard_append(data)
```

The `Cut callback` function will get whatever is in the textvar of the proper widget, paste it into the clipboard, and then empty the textvar. As you can see, you need to be careful to follow the order of the steps:

```
def on_pop_Cut(*args):
    print('menudemo_support.on_pop_Cut')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    which = args[0]
    if which == 1:
        data = _w1.EntryData.get()
    else:
        data = _w1.EntryData2.get()
    root.clipboard_clear()
    root.clipboard_append(data)
```

```
    if which == 1:
        _w1.EntryData.set('')
    else:
        _w1.EntryData2.set('')
```

The Paste function pulls the data from the clipboard and places it into the proper Entry widgets' **textvar**. We use the **clipboard_get()** function to do this and the variable **cbdata** holds this value until we put it into the **textvar**:

```
def on_pop_Paste(*args):
    print('menudemo_support.on_pop_Paste')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    # Get data from Entry widget and paste it to the clipboard
    which = args[0]
    cbdata = root.clipboard_get()
    if which == 1:
        _w1.EntryData.set(cbdata)
    else:
        _w1.EntryData2.set(cbdata)
```

The Quit function of the popup menu doesn't care about which of the Entry widgets it is being called from. It just calls **sys.exit()**:

```
def on_pop_Quit(*args):
    print('menudemo_support.on_pop_Quit')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    sys.exit()
```

That ends the popup menu section and now we only have the Button bar Menu to deal with. Now would be a good time to save your code module, just in case.

Just like with the main menu callback functions, we will display a popup message box for each function. We can copy the last three lines of one of the button bar callbacks and paste them into the next one, changing what needs to be changed:

```
def on_btnClear(*args):
    print('menudemo_support.on_btnClear')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Clear Button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The next callback function is the `btnCut` function. The only change in the last three lines would be the message. The title and `messagebox` call is the same as the `btnClear` function:

```
def on_btnCut(*args):
    print('menudemo_support.on_btnCut')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Cut button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The `btnFileNew` function is next. Again, the only line that is changed from the `btnCut` function is the message line:

```
def on_btnFileNew(*args):
    print('menudemo_support.on_btnFileNew')
    for arg in args:
```

```
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the New button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

`FileOpen` is the next callback. Again, the message is the only thing that changes.

```
def on_btnFileOpen(*args):
    print('menudemo_support.on_btnFileOpen')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Open button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The `FileSave` and `FileSaveAs` are the next two callbacks:

```
def on_btnFileSave(*args):
    print('menudemo_support.on_btnFileSave')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Save button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)

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

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



```
    titl = "Button Menu"
    msg = "You clicked the File Save As button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The **btnHelp** and **btnInfo** callbacks are next and just like the last few callbacks, the only thing that changed is the message lines:

```
def on_btnHelp(*args):
    print('menudemo_support.on_btnHelp')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Help button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)

def on_btnInfo(*args):
    print('menudemo_support.on_btnInfo')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Info (About) button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

The last two are the **btnPaste** and **btnCopy** callbacks:

```
def on_btnPaste(*args):
    print('menudemo_support.on_btnPaste')
    for arg in args:
        print('    another arg:', arg)
```

```
    sys.stdout.flush()
    titl = "Button Menu"
```



```
    msg = "You clicked the Paste button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)

def on_btn_Copy(*args):
    print('menudemo_support.on_btn_Copy')
    for arg in args:
        print('    another arg:', arg)
    sys.stdout.flush()
    titl = "Button Menu"
    msg = "You clicked the Copy button"
    messagebox.showinfo(titl, msg, icon=messagebox.INFO, parent=_top1)
```

That is it. Now, save the code file.

Conclusion

You can see that, while menus of any type are simple concepts, they can be very powerful in providing options to the users of your programs. Menu bars can provide the concept of quick jumps to various parts of your program, context menus can make the program easier to use and button bars can provide a more modern look while still giving most of the menu bar capabilities.

That is it for Menus, Popup menus, and Button bar menus. In our next chapter, we will look at using the ttk widgets.

We concentrated on the <Button-3> event, but there are many others. To find out more about events, please refer to the New Mexico Tech Tkinter 8.5 reference at **https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html**.

Multiple Choice Questions

1 To bind the mouse-3 click event to a callback in code, you would use:

   a. widget.bind(Button-3, popupmenu.callbackname)

   b. widget.bind('<Button-3>', lambda e: menudemo.popup1(e, 2))

c. widget.bind('<ButtonRelease-3>', lambda e: menudemo.popup1(e, 2))

2. Menus and PopupMenus both use the same editor.
   a. True
   b. False
3. The types of menu items are:
   a. Command, Cascade, and Separator
   b. Command, Cascade, Separator, Clear, and Graphic
   c. Command, Cascade, Separator, Radiobutton, and Checkbutton

Answers

1. Both B and C are correct. A is incorrect because the event (the first portion of the definition) must be presented as a string and you must enclose the event name in "<" and ">".

   B is correct and is used directly from our code. C is also correct since that event is triggered when Mouse Button-3 is released. Sometimes, this can keep multiple events being raised due to "mouse button bounce".
2. True
3. C

Questions

1. When creating a menu keyboard accelerator, what do you have to do?
2. Can you break a long list of command items in a menu?
3. Can menus have images?

Answers to the questions

1. First, you must use the accelerator attribute to set the text that should be shown next to the menu item something like this "(*Ctrl + N*)". Then, bind the keystroke to the event you wish to raise.
2. Yes. Set the `columnbreak` attribute to 1 for the item that you want to be the first in the new column. And Yes, you can have more than two columns if needed.
3. Yes. Use the image attribute for the item. Both command and cascade menu items can have images as well as `Radiobuttons` and `Checkbuttons`.

Keywords

- Command
- Cascade
- Separator
- Radiobutton
- Checkbutton
- Accelerator
- Binding
- Event
- Callback
- Popup menu
- Tooltip
- Column Break.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7
Using ttk Widgets

Introduction

In *Chapter 3, Standard TK Widgets*, we introduced the Tk widgets supported by PAGE. PAGE supports 20 ttk widgets, most of which are "cousins" of Tk widgets. The ttk widgets all start with "T" and can be found in the Toolbox under the Themed Widgets section.

The ttk widget set only contains 6 widgets that do not have an analogue in the Tk widget set. These widgets are TCombobox, TNotebook, TProgressbar, TSeparator, TSizegrip, Treeview (and to a certain extent the PNotebook).

The biggest difference that you will notice is the number of attributes that are available to control the look and feel of the widgets. For example, the tk Button has

30 attributes in the Attribute Editor that you can use to control the Button widget. Compare that to the ttk TButton which only has 11 attributes. Most ttk widgets don't give you the ability to change the background color, foreground color, the font or much else directly from the Attribute Editor.

In this chapter, we will examine why one would use these over the standard tk widgets.

What you will learn

In this chapter, you will learn the following topics:

- What is a ttk widget?
- Differences between the ttk widgets and the tk widgets
- What is a Theme?
- What is a Style?

What is a ttk widget

Ttk stands for Themed Tk widget. Most of the ttk widgets are based directly on "normal" Tk widgets, with many of the attributes that control the "look and feel" of the widget no longer easily accessible directly from PAGE. You probably are asking why anyone would do this and why would anyone want to use such a widget with all the restrictions.

The whole idea behind Themed widgets came about when the Tk library was ported to operating systems like Microsoft Windows and Mac and got ported to programming languages like Python and others. The Tk widgets, while free and available in many languages, started getting a bad reputation because the widgets all looked "old school" and did not fit into the new look of the more modern operating system desktops. So, a concerted effort was made to make all the widgets "fit in" with the modern desktop. Each of the widgets was given a "facelift" and that facelift was called a style. Then, each of the styles was gathered into a theme. When the theme was enabled, all the widgets had, in theory, a consistent look and feel.

Originally, there were just four themes available. They were called default, clam, alt, and classic. More were added as time when on, some for Mac OS X and others for Microsoft Windows. Those newer themes were (and are still) specific to the operating system and are not available on any other operating systems.

What is a theme?

As mentioned, a theme is a collection of styles for various ttk widgets and there are various themes available for different operating systems.

Operating System	Themes							
	clam	alt	default	classic	aqua	winnative	vista	xpnative
Linux	X	X	*	X				
Mac OS X	X	X	X	X	*			
Windows	X	X	X	X		*	X	X

Table 7.1: Themes by operating systems

In *Table 7.1*, you will see a list of the standard themes by an operating system. An "X" means that the theme is available for the operating system, and a "*" means that the theme is the default one.

When you are designing your GUI, the ttk widgets will use the default for the operating system you are using. In PAGE 7.6 and before, you can only see the widgets in the default theme for your operating system. For example, if your operating system is Windows, the widgets will use the **winnative** theme.

When creating a GUI for your program using any ttk widgets, remember that if you plan to run the program on an operating system that is not what you designed it on, be sure to use one of the four "global" themes. For example, you designed your program on Windows and you want to share it with someone who runs Linux. The theme that you designed in under will be the winnative theme, which Linux can't use. So be sure to use either the clam, alt, default, or classic themes and your program will look the same on any operating system.

What is a style?

A style is the actual look and feel of individual widgets. This includes colors, fonts, relief, and more. Basically, the things that you can't control through the PAGE Attribute Editor for the various ttk widgets that you can set for Tk widgets.

Styles are based on the concept of elements. Normally, each ttk widget style is made up of one or more elements. The group of elements is called a layout. Let us take a look at one of the simplest ttk widgets, the ttk Button. The Button has a class name of **Button** and the style of this widget class is called **TButton**. Every ttk Button has the TButton style by default. Every theme creates its own TButton class.

A typical ttk Button has four elements. *Figure 7.1* shows the elements for the TButton style when using the default theme. Think of the widget itself as an empty box that holds the elements.

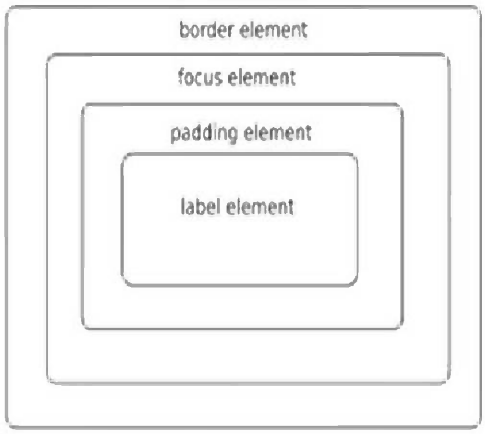


Figure 7.1: TButton Elements

In this example, the first element or outer element is the border element. You can consider this the parent element. The next element is the focus element and child of the border element. The focus element has a child named padding and that element has the final child element named label. The label element holds the text for the widget, the image for the widget or both.

Each element has a "sticky" attribute that defines which of the sides of the box (or parent) the element will stick to. There are four sides to each element named for the compass directions. North, east, south, and west and they use the first letter of the directions. So, if, for example, an element has a sticky attribute of "ew", the element will expand right and left to the edge of the parent. In this case, there are no north or south expansions defined, so the element layer won't expand to the top and bottom of the parent.

A ttk style does not have to have a layout.

If we were to open Python in the interactive mode, we can examine the different layouts. Here is what the layout for the TButton in the default theme looks like:

```
>>> import tkinter as tk

>>> import tkinter.ttk as ttk

>>> style=ttk.Style()

>>> style.layout('TButton')

[('Button.border', {'sticky': 'nswe', 'border': '1', 'children': [('Button.
focus', {'sticky': 'nswe', 'children': [('Button.padding', {'sticky':
'nswe', 'children': [('Button.label', {'sticky': 'nswe'})])])])])]
```

```
>>>
```

When a widget class has a layout, it will be returned as a list of tuples which have the element name and a dictionary that describes that element.

The outermost element is the border. It has its sticky attribute which is, in this case, north, south, east, and west (so the border will expand to all sides of the widget) and a border attribute that has a width of 1 pixel. It also has a child element. In this case, it's the focus element. It has a sticky attribute that also expands north, south, east, and west. The focus element does not have a border, but it does have a child, which is the padding element. That has only one attribute (as you might guess) the sticky attribute and again it's north, south, east, and west and it ALSO has a child element, the label element. It has a sticky attribute as well and, yet again, it is 'nsew'.

Each of the elements has options that you can examine by using the `element_option` method. Here is a dump of all the `element_options` for the TButton Elements:

```
>>> style.element_options('Button.border')

('background', 'borderwidth', 'relief')

>>> style.element_options('Button.focus')

('focuscolor', 'focusthickness')

>>> style.element_options('Button.padding')

('padding', 'relief', 'shiftrelief')

>>> style.element_options('Button.label')

('compound', 'space', 'text', 'font', 'foreground', 'underline', 'width',
'anchor', 'justify', 'wraplength', 'embossed', 'image', 'stipple',
'background')

>>>
```

To find out the value of the attributes for a particular option, you can use "`style.lookup(element,option)`". So, for example, we wanted to find out the background attribute of the Button border, we can use:

```
>>> style.lookup('Button.border','background')

'#d9d9d9'
```

Creating a simple custom style

We can easily create our own styles if there is nothing that suits our needs in any of the available themes. When I'm setting up a new style, I usually start with the **default** theme, not just the default for the operating system.

We will create a quick program to demonstrate the power of creating a custom style for a TButton widget.

Start PAGE and place a TButton and a regular Tk Button on the form. Set the command attribute for the Tk Button to "`on_btnMyStyle`".

Save your project and generate the GUI and Support Python modules:

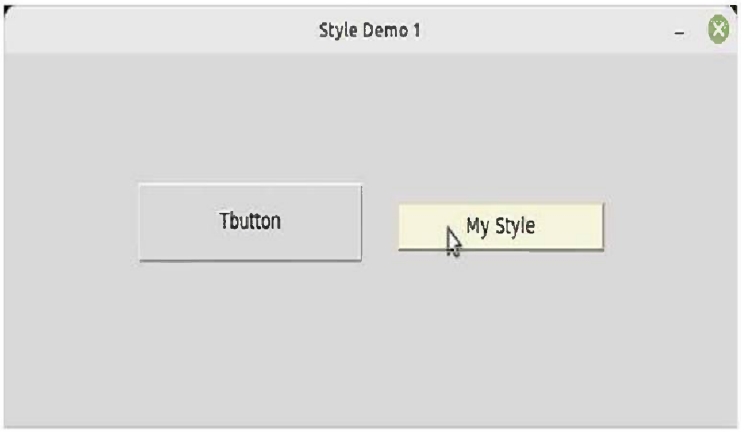


Figure 7.2: TButton Custom Style

This is what your GUI should look like. Notice that the TButton looks somewhat flat and bland. We will fix that.

In the Support module, you need to modify the main function to include a call to the startup function just before the final line:

```
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 = simplestyle.Toplevel1(_top1)

    startup()

    root.mainloop()
```

Next, create the startup function with the following code:

```
def startup():

    style = ttk.Style()

    my_styles(style)

    global plain

    plain = True
```

We first need to create an instance of the **ttk.Style** object, then we will call the function **my_styles()** and pass that object as a parameter. We will also create a global variable called "plain" and set it to True. This will be used to change the style from our custom style back to the normal bland style.

Now, we can create our **my_styles()** function. The first thing we do is create what is called a map that holds the background and foreground color sets for the various states (in this case, if the widget is disabled or if the mouse cursor is hovering over the widget, which triggers the **active** state. Notice that I have called the map dictionary **my.TButton**. This will only change the mapping for a single TButton widget. This actually could be any number of buttons, but we are doing it this way, so we can re-apply the default style later on to 'un-style' the widget later on:

```
def my_styles(sty):

    sty.map('my.TButton',

          background=[("disabled", "#d9d9d9"), ("active", "seagreen2")],

           foreground=[

               ("disabled", "gray45"),

               ("active", "gray6"),
```

```
        ])
```


Now, we can use the configure method to set the various things we want to change, like borderwidth, background, and foreground colors to what we want it to be. Again, we have named it `my_TButton`:

```
    sty.configure(
        "my.TButton",
        foreground="white",
        background="cornflowerblue",
        relief=RAISED,
        borderwidth=3,
        padding=[4, 4, 4, 4],
        font="Ubuntu 12 bold",
    )
```

Finally, we need to have a way to set and clear the style for the TButton. We'll do that in the callback function of the Tk Button:

```
def on_btnMyStyle(*args):
    if _debug:
        print('simplestyle_support.on_btnMyStyle')
        for arg in args:
            print('    another arg:', arg)
        sys.stdout.flush()
    global plain
    if plain:
       _w1.TButton1.configure(text="My TButton Style", style="my.TButton")
        _w1.btnMyStyle.configure(text="Remove My Style")
        plain = False
    else:
```

```
    _w1.TButton1.configure(text="My TButton Style", style="TButton")
    _w1.btnMyStyle.configure(text="Apply My Style")
    plain = True
```


We first, declare that we will be using the **plain** global variable then we check to see whether it is **True** or **False**. In our startup function, we set it to True. Now, when the user clicks on the Tk Button for the first time, we configure the TButton by setting the text and setting the style attribute to our custom style. We also change the text of the Tk button to "`Remove My Style`

When the user clicks on the Tk Button again, plain will be False, so we go through the configuration again; this time setting the style to the "normal" TButton style, then set the text of the Tk Button to "`Apply My Style`" and finally setting the plain global back to True:

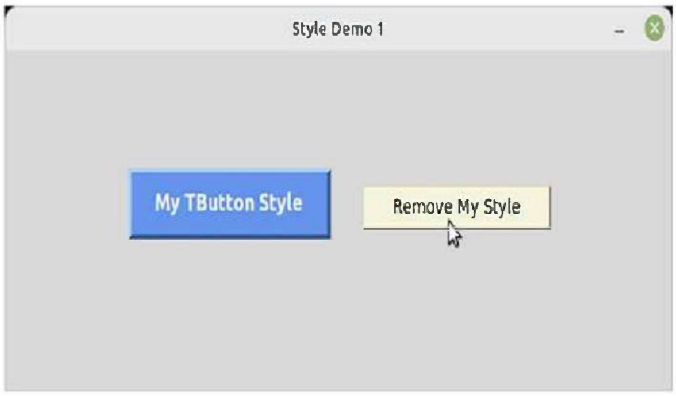


Figure 7.3: TButton with Style set

The code for this project is provided in the code folder and is named `simplestyle.py`

Getting more themes

You are not just locked into the few themes that are available for your operating system or creating your own. You can find more on the Internet. Here are a few links to get you started.

The best list of third-party themes that I've found can be found at:

https://wiki.tcl-lang.org/page/List+of+ttk+Themes

There is a very nice package called ttkthemes which can easily be used in Python and Tkinter

and Tkinter.

https://github.com/TkinterEP/ttkthemes/tree/master/ttkthemes/themes

A very nice theme is named Waldorf. You can find it at **https://wiki.tcl-lang.org/page/waldorf+ttk+theme.**

There is a very nice theme called Sun-Valley. You can find it at **https://github.com/rdbende/Sun-Valley-ttk-theme.** The author also has two other theme sets called Forest and Azure. All three theme sets support both a light and dark set.

Special ttk widgets

As I stated at the beginning of the chapter, there are 6 widgets that are not simply analogs of the standard Tk widgets. We will briefly examine them in this section.

TCombobox widget

The TCombobox widget is a very useful widget that combines a Tk Entry widget with a Tk Listbox that drops down on demand all in one. The setup is a bit complicated but is one that gets used very often:

Figure 7.4: TCombobox

The easiest way to configure the `TCombobox` is in your support module after you set a few attributes in PAGE. You should always set the Alias to a unique value, set the export selection to No, so that the selection is not copied to the clipboard, and then make sure the text variable (text var) is set to a unique value. PAGE provides a `combobox` and a default value, but if you have more than one `TCombobox` in your project (even if they are on different forms) you will have a conflict. Then, in your support module, you must create a list of all the options that your `TCombobox` will offer. Next, you set the values attribute with that list. After that, you need to bind the widget to the virtual event "`<<ComboboxSelected>>`" and finally, you must create the callback function that will be used when the user clicks on one of the options:

```
options = ["One", "Two", "Three", "Four", "Five"]

_w1.TCombobox1["values"] = options
```

```
_w1.TCombobox1.bind("<<ComboboxSelected>>", lambda e: on_ComboSelect(e))
```

Once these are set up, then you need to create the callback function. Here, I've chosen the name "`on_ComboSelect`" for the name of the callback:

```
def on_ComboSelect(e):

    selected = _w1.combobox.get()

    print(f"Combobox Select: {selected}")
```


TNotebook

The TNotebook widget is a super container for other widgets with different pages or tabs. You can have many pages all holding multiple widgets:

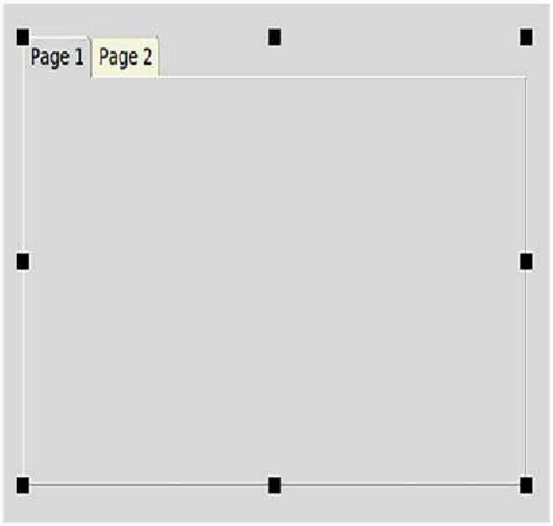


Figure 7.5: TNotebook

You can take a look at the **WidgetDemo** program that I created that comes with every distribution of PAGE. You can find it in the `examples|widgetdemo` folder of your PAGE distribution.

TProgressbar

There are times that your program will need to do something like download a file from the Internet or query a database that takes time. Good programming practices suggest that you as the programmer, provide some visual indication that something is going on, so the user does not think that the program is frozen:

Figure 7.6: TProgressbar

The TProgressbar is just the widget for that. It is simply a bar that slides in a trough to show a status. If you know how far you are into the process (as a percentage of being done), you can send that percentage into the TProgressbar after you have set the mode to "**determinate**". If you do not know how long the process will take, you can set the mode to "**indeterminate**" and use the `.start()` method to let the bar free

run within the trough. You can stop it again with the `.stop()` method. Again, you can look at the **Widget**Demo code to see how these modes work.

TSeparator

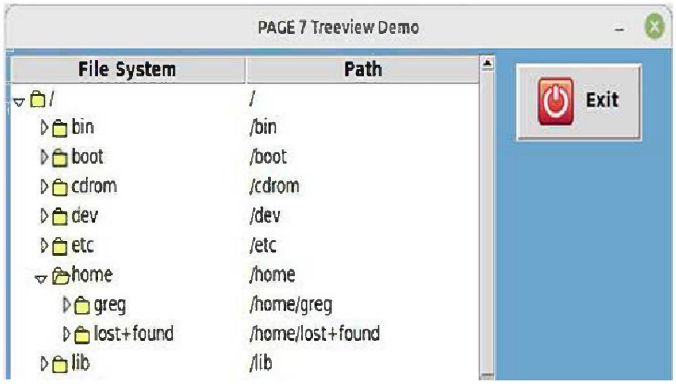
The TSeparators come in two "flavors": vertical and horizontal. They simply provide a line that you can use to separate groups of widgets.

TSizegrip

If you have programmed your GUI in the Relative mode, the user can resize the form by dragging one corner. The TSizegrip gives an easy visual indicator that the program form can be resized. If you programmed the GUI in the Absolute mode, the form cannot be resized, so do not use this widget.

Treeview

The Treeview widget is a great widget to present data in either a hierarchical manner, like a file manager does or in a quasi-spreadsheet manner (sometimes called a multi-column list). It is a very complicated widget to use, but I have provided two demos that are available in the examples folder of your PAGE distribution:



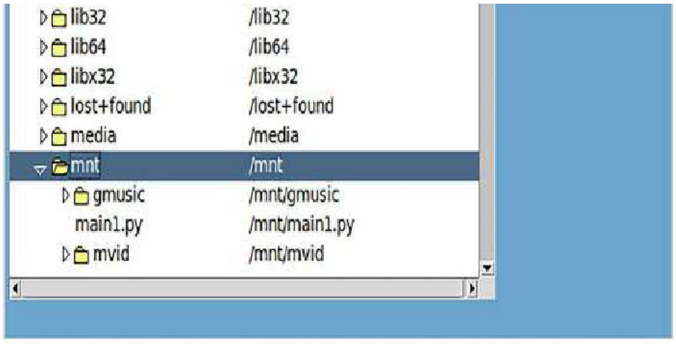


Figure 7.7: Treeview Widget

The WidgetDemo shows the spreadsheet-like mode and the demo in the Treeview folder shows a simplified file manager version. Both demos provide all the code that you should need to use this widget in your programs.

Conclusion

In this chapter, you learned the basics of the ttk Widget set, what themes are, what styles are, and how to create your custom styles. You have also learned about the ttk widgets that do not have a cousin in the Tk widget set, like the TCombobox and the TNotebook. I have also given you several links to get you started finding other third-party theme sets.

There is so much more I would love to cover in this chapter, but there is just so much space.

Multiple Choice Questions

1. Ttk widgets are just fancy versions of the standard Tk Widgets. State true or false.
   a. True
   b. False
   c. Sort of
2. PAGE allows you to see what the ttk widgets will look like in the designer mode. State True or False.
   a. True
   b. False
   c. Sort of

Answers

Answers

1. C – Sort of. Many of the ttk widgets are simply fancy-styled Tk widgets, but six additional widgets are provided that don't have Tk widget cousins.
2. C – Sort of. PAGE displays the ttk Widgets using the default theme for your operating system. Under Linux, this is the default theme. Under Windows, this is the winnative theme and under Mac OS X, it is the aqua theme.

Questions

1. What are the easiest steps to use a TCombobox in your own program?
2. Ttk widgets are hard to use and not worth the trouble.

Answers to the questions

1. Step 1: Place the ttk TCombobox on your form. Set the Alias, set export selection to No, and make sure you have a unique text variable (which PAGE sets by default to "combobox".

   Step 2. In your support module, create a list that contains the options you want the user to be able to pick from.

   Step 3. Assign the list to the values attribute of the TCombobox.

   Step 4. Bind the TCombobox to the "<<ComboboxSelected>>" event and set the callback function name.

   Step 5. Create the callback function to handle all the possible options.
2. In their native state, ttk widgets are actually easier to use than Tk widgets, since they have fewer attributes to worry about. However, if you want to make a ttk widget look different than any of the themes you have available, then the answer would be True, at least the first few times.

Key terms

- ttk
- Themes
- Styles
- Scrollbars

- Elements

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8
Custom Controls

Introduction

In the previous chapters, we covered almost all the widgets that PAGE supports, both Tk widgets and ttk widgets. However, sometimes all those widgets just do not offer you that one widget that would make your project usable enough. You might need a 'date picker' to make it easier for your users to select a date or a way to display some data in a spreadsheet-like widget instead of a widget like the ttk

Treeview widget. All these things and more are available to you through the Custom Widget placeholder.

In this chapter, you will learn not only how to use the Custom Control Widget, but also how to use some other third-party widgets in your own projects

We will look at three short projects in this chapter which will give you a good insight on using third-party libraries in your own projects.

What you will learn

In this chapter, you will learn the following topics:

- How to use the Custom Control Widget
- How to use a Custom date picker

- How to use a ScrolledCheckedListbox widget
- How to use the Plotext Library

Custom Control Widget

There is only one rule, and it is not a hard and fast rule, that you need to remember when using the Custom Widget placeholder widget. Always place the custom widget holder inside a Frame, either a Tk Frame or ttk Frame. Do not use a LabelFrame widget. It will work, but the Label portion is unnecessary and will only cause complications later on.

We will get a lot of information in this chapter. However, there is not enough space in this chapter to provide the full source code. The full code for all three projects is provided in the *Chapter 8 - Custom Controls* code folder. We will go through more examples and less full explanation of all the options and attributes and possible uses of just one example. We have tried to provide multiple comments in the example code to help you understand the ways to use the provided Widgets.

Project #1 – tkDatePicker

In Project #1, you will learn how to use the `tkDatePicker` widget. This has two modes: one that you need to use the Custom Widget holder in PAGE and the other one you can call from a standard Tk Button:

Figure 8.1: tkDatePicker

Before you start to use this widget, you must install it into your Python instance. So, open a terminal and type:

```
pip3 install tk-datepicker
```

or

```
pip install tk-datepicker
```

The difference between the two is how you start Python on your system. If you start Python with **python3**, use the first one. If you start Python with the simple **python**, use the second.

Designing the program

As you can see from *Figure 8.1*, the form is very simple. Two Tk Buttons are used: one Entry widget, six Labels, and a Custom widget placed inside a Tk Frame. We use a Frame to hold the Custom widget, thus making it easier to move and resize it.

Saving the Frame and Custom widget for last, place all your other widgets as you see in *Figure 8.1*, and then place the Frame. We will put it at x=352 and y=140 in the demo with a width of 190 and a height of 25. Finally, place a Custom widget (located near the bottom of the toolbox under the Enhanced Widgets section) in the Frame. Then, use the mouse-3 (right click for most people) to get the Context menu. Select *Widget* and then select *Fill Container*. That will make the Custom widget holder fit completely inside the frame.

You do not add the Custom **`tk-datepicker`** widget until you edit the Support module.

Now, there is a callback for the second Tk Button which needs to be added in the Command attribute of the Button, which is **`on_showCalendar`**. Set the **`Textvariable`** attribute of the Entry widget to **`PopupReturn`**. The two blank labels need to have their **`textvariables`** set as well. Set the top to **`lblPopupReturn`** and the bottom to **`lblPulldownReturn`**. Finally, add a command to the Exit button callback as **`on_btnExit`**. Save your project and generate your Python files. Then, you can close PAGE.

Project #1 Code

Working in the Support Module, the first thing that must be done is to import the **`tk-datepicker`** module. When there is an add-on module, it is recommended to wrap the import statement with a **`Try | Except`** statement. This way if we give the program to someone else, and forget to tell them that it needs the third-party library,

the program does not just close with no information to the user as to why. Here is an example of how you might do this:

```
try:
    from tkcalendar import Calendar, DateEntry
except ImportError:
    # If the library is not installed, show error message to user and
    abort.
    print("~" * 50)
    print('This program requires the third party library "tkcalendar".
    \n')
    print(
        ' You can install it by typing "pip install tk-datepicker" \n\
          in a terminal or command prompt.'
    )
    print("\n  Program ends.")
    print("~" * 50)
    sys.exit(1)
```

You can see that by doing the import this way, when you share your programs with others, you can save a bunch of frenzied calls and emails.

Now, add a line to the `main` function to call the startup function that you will be creating:

```
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 = datepicker.Main(_top1)

    startup()

    root.mainloop()
```

Now, you need to create your startup function for this project. The first thing that we will do is clear the two empty Label widgets using the `.set` method. Then, we need to create a Custom style for the datepicker pulldown widget. Normally, it looks like a `ttk Combobox`:

```
def startup():

    _w1.lblPulldownReturn.set("")

    _w1.lblPopupReturn.set("")

    global style

    style = ttk.Style()

    style.configure(

        "my.DateEntry",

        fieldbackground="light green",

        background="dark green",

        foreground="dark blue",

        arrowcolor="white",

    )
```

Now, we need to do some configuration for the calendar. Since I am located in the United States, I set the local to "`en_US`". Then, we set the date pattern to match that for the U.S. which is "**MM/dd/yyyy**". Next, set the `firstweekday` attribute to what your locale uses. In the U.S., it's Sunday. Many other countries use Monday as the first day of the week. Then, set the weekend days, which in the U.S. is Sunday and Saturday which ends up being a list of **[1,7]**. Finally, set the style to `my.DateEntry` and set the binding to the virtual event `<<DateEntrySelected>>` and the callback for the event to `on_calSelect`. In the provided code, we have put detailed comments to help you:

```
    # If you wish to set the date locale and pattern, this MUST be done
    before
```

```
    # setting the firstweekday and weekenddays attributes...
    _w1.Custom1.configure(locale="en_US")
    _w1.Custom1.configure(date_pattern="MM/dd/yyyy")
    # For locations that use Sunday as the first day of the week, otherwise
    # firstweekday='monday'
    _w1.Custom1.configure(firstweekday="sunday")
    # Weekenddays in U.S.A are Sunday and Saturday.  For other locations,
    # the weekenddays would be [6, 7]
    _w1.Custom1.configure(weekenddays=[1, 7])
    # Set the ttk style for the control
    _w1.Custom1.configure(style="my.DateEntry")
    # Bind the callback
    _w1.Custom1.bind("<<DateEntrySelected>>", on_calSelect)
```

The next thing that you should do is creating the callback for the binding we just programmed:

```
def on_calSelect(param):
    print(f"Calendar Select: {_w1.Custom1.get_date()}")
    global PullDownDateReturn
    PullDownDateReturn = _w1.Custom1.get_date()
    _w1.lblPulldownReturn.set(PullDownDateReturn.strftime("%m/%d/%Y"))
```

Next, you need to create the callback code for the Show Calendar button. The skeleton for this was created by PAGE. In this callback function, we will start with a small, embedded function inside the callback function:

```
def on_showCalendar(*args):
    if _debug:
        print("datepicker_support.on_showCalendar")
        for arg in args:
```

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

The embedded function is a simple function that will set the selected date information into the Entry widget and the Label using the `.set` function:

```
    def print_sel():
        dt = cal.selection_get()
        frmtd = dt.strftime("%m/%d/%Y")
        # selection_get() returns a datetime.date object
        _w1.PopupReturn.set(frmtd)
        _w1.lblPopupReturn.set(frmtd)
```

Now, we need to make sure that the datetime module is imported, get the current date, and then we will create the actual tkinter object in the code, which will become our popup calendar. You will code the initialization routine pretty much the same way that we did with the preceding pulldown version:

```
    import datetime
    today = datetime.date.today()
    top = tk.Toplevel(root)
    cal = Calendar(
        top,
        font="Arial 14",
        selectmode="day",
        locale="en_US",
        firstweekday="sunday",
        weekenddays=[1, 7],
    )

    cal.pack(fill="both", expand=True)
    ttk.Button(top, text="ok", command=print_sel).pack()
```

Finally, you need to look for a line that starts with **Custom** This is added by PAGE and should be very close to the end of the Support module. You will need to change this to:

```
Custom = DateEntry
```

Don't forget to add the **sys.exit()** to the Exit button callback. Save your file and test your project.

When using the pop-up calendar option, we provided all the initialization code needed in the callback of the Tk Button that "calls" the pop-up calendar. While this seems like overkill because the initialization code is called every time the user clicks on the button for the pop-up version, this version of the calendar is basically created "on the fly" each time the user asks for the widget to be shown. In the case of the pulldown version (inside the Custom Widget holder), we set all the initialization information in the startup function and that is only called once.

Normally, you would only use the **popup** calendar (the one called from the Tk Button) or the **pulldown** calendar (the one using the Custom Widget), not in the same project. Both are included to show you how to program them both. Personally, I would stick with the pulldown version.

Project #2 – ScrolledCheckedListbox

Several years ago, I was writing a program using PAGE that was going to be a front-end database program, which is a Recipe database. One of the functions of the program needed to be able to relate data from one database table with another and show selected recipe categories for a particular recipe. For example, if the recipe was one that contained chicken and was a slow cooker recipe, I wanted to be able to show and enter these categories by simply using checkboxes. If a box was checked, the recipe would include that category in the database. Later, if I wanted to see all the recipes that include chicken, an SQL statement could be created that would filter and show only the recipes that included chicken as an ingredient.

Unfortunately, PAGE and Tkinter do not include a widget that can create a Listbox of CheckButtons that can be scrolled up and down, which is what I felt I needed. I either had to change the way my program would work or create a "Metawidget" (a widget that is based on multiple other widgets like the combobox) of my own. From that need, came the ScrolledCheckedListbox widget:

Figure 8.2: ScrolledCheckedListbox

The Metawidget that I created does exactly what I wanted and it looks just like a 'normal' Tkinter widget. You will see that it is easy to include in your programs and also easy to customize for your own needs. The only things you need are the Custom Widget holder from PAGE, the widget code module and some data to display. You don't need to install it, just include it in the project folder and import it into your program.

Designing the program

Figure 8.3 will give you an idea of what the demo program looks like:

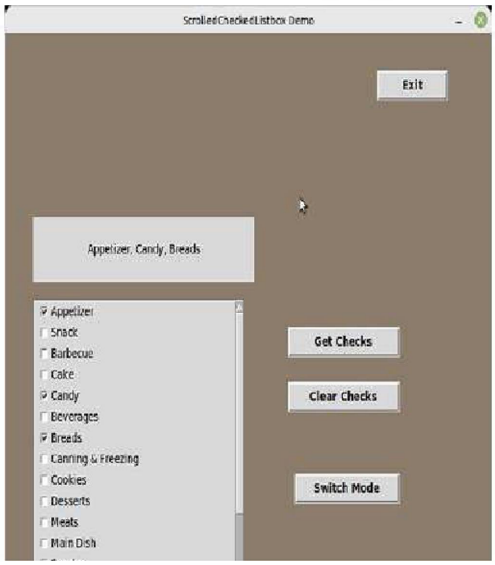


Figure 8.3: ScrolledCheckedListbox Demo

You can see there are just four tk Buttons, one Message widget, and our Custom Widget. As usual, since we are using the Custom widget holder, we also have a frame that acts as a container for it. The Message widget acts as a multi-line label to display all the categories that are checked and dynamically updated when a category is selected or unselected.

Simply, place your four Tk Buttons and the Message widget where you want them. Place the **Frame** and set the **Width** to **305** and the **Height** to **345**. Drop a **Custom Widget** holder into the Frame as we did in the first project. Again, use the mouse-3 (right click) to access the Custom Widget Context menu and select the **Widget | Fill Container** option. For your **Exit** button, set the command to **on_btnExit**; for the **Get Checks** button, set the command to **on_btnGetChecks**, the **Clear Checks** button to **on_btnClearChecks**, and set the **Switch Mode** button to **on_btnSwitchMode**.

Save your project and generate the Python files.

Project #2 Code

In your support module, add the following line in the import section:

```
from ScrolledCheckedListBox import ScrolledCheckedListBox
```

You can use the Try | Except statement method as I showed in Project 1 if you want.

Be sure to add the call to the startup function next to the last line of the **main** function.

Now, for the actual startup function, we will create two lists as globals that will hold the categories. The first is a single list and the second is a list of lists holding both the text and a "record" number. That way when the user makes a selection, the category record number is also included. A short example is shown for each list:

```
def startup():
    global ListInfo, ListInfo2
    # ======================================================
    # Create the list of text items for the ScrolledCheckedListBox
    # ======================================================
    ListInfo = [
        "Appetizer",
        "Snack",
```

```
        "Barbecue",
        ...
        "Mexican",
        "Asian",
    ]
    ListInfo2 = [
        ["Appetizer", 1],
        ["Snack", 2],
        ["Barbecue", 3],
        ...
        ["Cereal", 18],
        ["Salads & Dressings", 19],
        ["Sauces", 20],
    ]
    global listmode
    listmode = 1
    initialize_custom_widget()
```

We also need to set a global that holds which list option to use. Normally, you would use only one or the other and most likely it would be listmode 2, since that is the most useful. Finally, we call the initialization function for the custom widget.

In the initialization function, we first set the callback for when the user selects one of the options (Checkbuttons) using the **cback** method of the Custom widget, then call the load method for the Custom control to load the list of options (which creates each of the Checkbuttons. At the end of the function, we clear the Message widget and finally, we set a few Checkbuttons as an example and a starting point:

```
def initialize_custom_widget():
    global listmode
    _w1.Custom1.cback = on_SCLB_click
    if listmode == 1:
```

```
        _w1.Custom1.load(ListInfo)
    else:
        _w1.Custom1.load(ListInfo2)
    clear_label()
    set_labels()
```

Next we create the callback function.

```
def on_SCLB_click(s=None):
    update_label()
    print(_w1.Custom1.get())
```

The next function we create will be the `set_labels()` function. This uses the set method of the Custom widget to select the various Checkbuttons we want (in this case, it is three in a list, but normally, it is simply a single option at a time), then we call the update label to get the list of the selected buttons, and display the list in the Message widget:

```
def set_labels():
    _w1.Custom1.set(["Appetizer", "Candy", "Breads"])
    update_label()
```

Now, we need to create the `update_label` function. While the code looks complicated at first, it's simply calling the get method of the widget which returns a list of the text (and if in mode 2, the second column of data) of the listbox widgets that are checked. The code then walks through the returned list, and checks to see whether the data is simply an item in a list or in the case of mode 2, a list itself, separates the two columns into discrete data elements, and appends the important information (the text of the selected item) into another list. Finally, the items of the list are separated by commas to separate items and sent to the Message widget for display:

```
def update_label():
    dat = _w1.Custom1.get()
    lst = []
    for x in dat:
        # pprint(len(x), x)
```

```
        # print(len(x),x)
```



```
        if len(x) == 2:
            t = x[0]
            k = x[1]
            lst.append(t)
        else:
            lst.append(x)
    s = ", ".join(lst)
    _w1.Message1.configure(text=s)
```

The **clear_label** function simply replaces the text within the Message widget by setting the text attribute to an empty string. (You can always use the textvar property of the Message widget, but I wanted to show another way to set the text.):

```
def clear_label():
    _w1.Message1.configure(text="")
```

The next support function will clear all the currently checked options in the widget. It simply calls the clear method of the widget and then sets the message box text to an empty string:

```
def on_btnClearChecks(*args):
    if _debug:
        print("Custom1Demo1_support.on_btnClearChecks")
        for arg in args:
            print("    another arg:", arg)
        sys.stdout.flush()
    _w1.Custom1.clear()
    _w1.Message1.configure(text="")
```

The **on_btnGetChecks** callback function will use the get method of the widget (which comes back as a list), update the Message box widget, and show a popup **messagebox** through the **showinfo** message call:

```
def on_btnGetChecks(*args):
```

```
if _debug:
```

```
        print("Custom1Demo1_support.on_btnGetChecks")
        for arg in args:
            print("    another arg:", arg)
        sys.stdout.flush()
    lst = _w1.Custom1.get()
    print(lst)
    titl = "ScrolledCheckedListbox Demo"
    msg = f"Selected items: \n{lst}"
    messagebox.showinfo(titl, msg, parent=_top1, icon=messagebox.INFO)
```

The final callback we will examine is the **on_btnSwitchMode** callback. This is simply provided to switch between the straight list mode and the multi-column mode that not only holds the list of options for the widget, but also carries a second field which can be used to hold data like a database record key. A popup message box displays which list type is being used when the button is clicked:

```
def on_btnSwitchMode(*args):
    if _debug:
        print("scldemo_support.on_btnSwitchMode")
        for arg in args:
            print("    another arg:", arg)
        sys.stdout.flush()
    global listmode
    if listmode == 1:
        listmode = 2
        _w1.Custom1.load(ListInfo2)
    else:
        listmode = 1
```

```
        _w1.Custom1.load(ListInfo)
    titl = "ScrolledCheckedListbox Demo"
    if listmode == 1:
        msg = "Now running in single column data mode"
    else:
        msg = "Now running in MULTI column data mode"
    messagebox.showinfo(titl, msg, parent=_top1, icon=messagebox.INFO)
```

Last but not least, we change the line of code that PAGE created for us setting the variable "Custom" to the widget we want to use for this Custom Widget holder. We did this in the first project in this chapter. It usually shows up as one of the last few lines in the support file:

```
Custom = ScrolledCheckedListBox
```

That is it for the `ScrolledCheckedListbox` demo. If you want to know more about creating your own Metawidget, you can examine my code in the `ScrolledCheckedListBox` module that is included in the source code folder.

Project #3 – Plotext

There will be times that you have some kind of data that you will want to show as a plot or graph. You could use the matplotlib library, which is pretty much the standard library for plotting and graphs. However, that comes with a lot of overhead and the plot comes up in it its own window. What if you want to embed a plot into your GUI directly?

Plotext is a third-party library that was designed to plot directly to the terminal output. Originally, I was able to get Plotext to work within a PAGE form by using a ScrolledTextbox widget, but after working with the author of Plotext, he modified it to work directly plotting to a Label widget. The Plotext library is capable of providing colors for each different type of plot data, but because the Tkinter Label widget only supports one color at a time, we cannot use that capability. It is a trade-off between capability and simplicity of code. You can use different markers to denote the different data in the plots (see the multi-bar plot option of the demo) as shown in *Figure 8.4*:

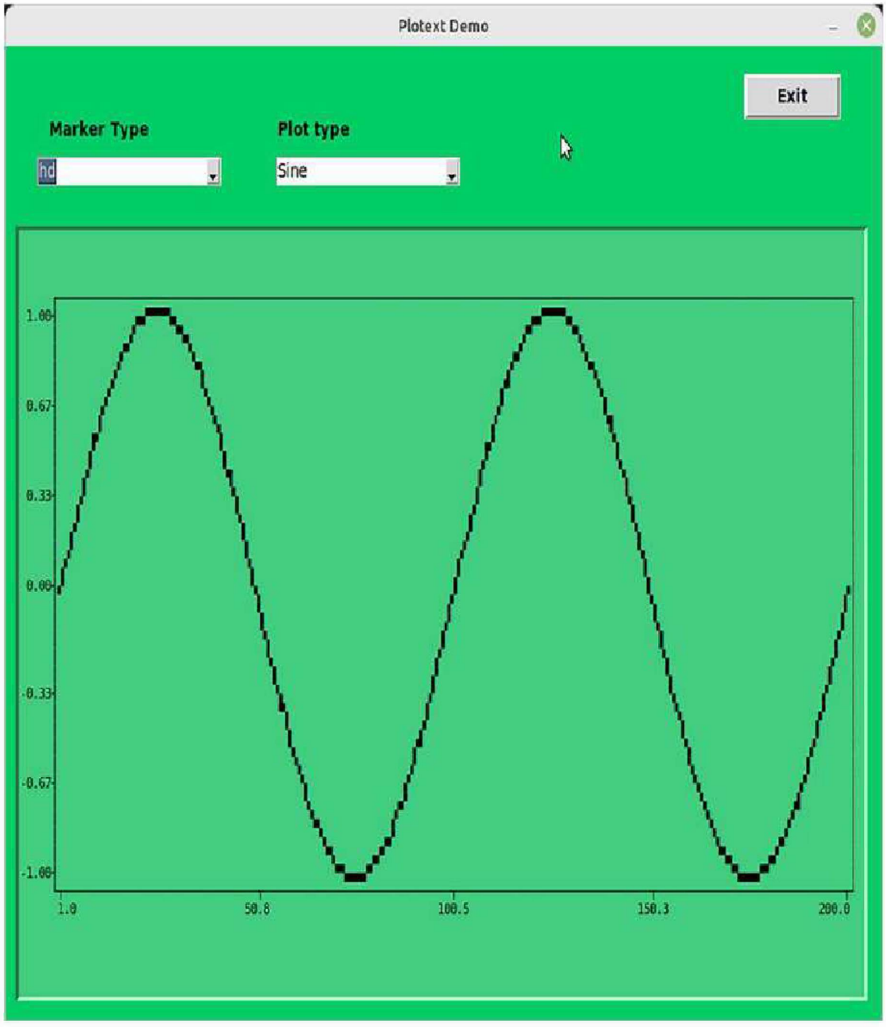


Figure 8.4: Plotext

Of course, you need to install `Plotext` in your Python installation by using pip or pip3:

```
pip3 install plotext
```

or

```
pip install plotext
```

As seen in *Figure 8.4*, it is very easy to make simple plots like a sine wave. However, more complex plots and graphs are also possible and not much more difficult.

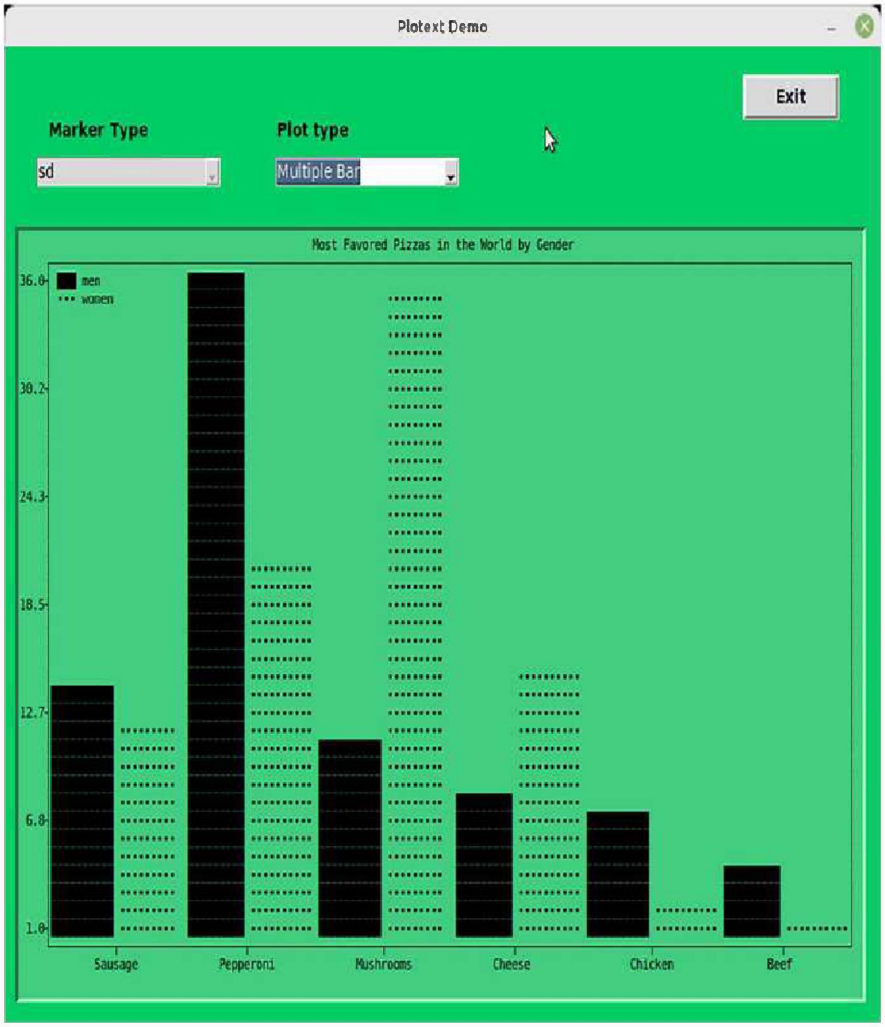


Figure 8.5: Plotext Multi-bar Plot

In the demo, I decided on supporting seven different plot types, and you will see that it is still easy to create much more useful plots and graphs that are embedded in a PAGE form.

During the Covid-19 crisis, I was actively monitoring the data for the United States and that is when I found the Plotext library. It was fairly easy for me to take the data from my spreadsheets and move them to a PAGE program that used Plotext to show the data:

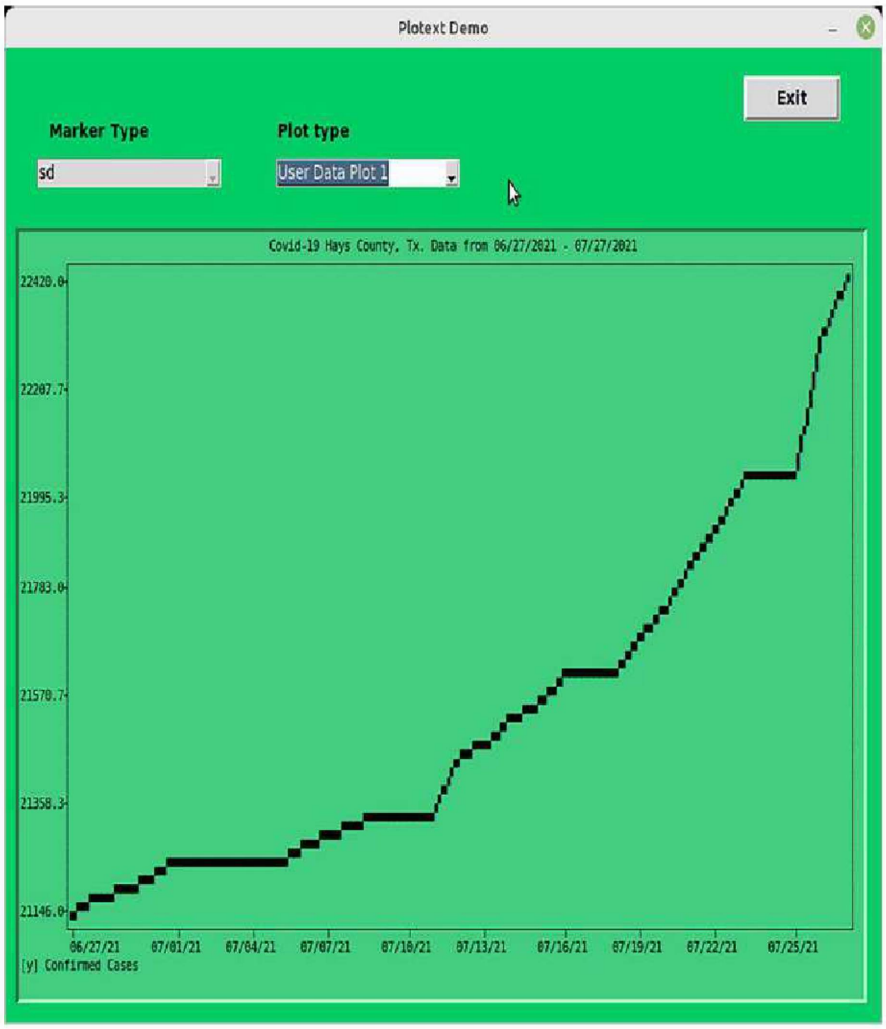


Figure 8.6: Covid-19 Data in Plotext

So, let us dig into how to create the demo for Plotext.

Designing the program

The form for this project is very simple. As you can see from the three figures in this section, there is one Tk Button, two TComboboxes, two "static" Tk Labels, and one Tk Label widget that act and display the "canvas" for the plot.

The TCombobox on the left provides the options for the Marker type that Plotext will use to create the chart for the simple Sine type. The TCombobox on the right provides a way to select what type of Plot will be used.

The form for the demo is **843** pixels wide and **705** pixels high. Of course, you can see that the majority of the space on the form is taken up with the Label that shows the

Plot. It is **561** pixels high and **819** pixels wide. You can make it pretty much any size you want since Plotext provides a method that controls the size of the plot called (strangely enough) **plotsize**.

You can even use some of the tricks we learned to create a second form that pops up to show the plot if you want to. The bottom line is that you get to control what kind of plot and where the plot data will show up.

Project #3 Code

There is a great deal of code in this project; mostly because of the Covid-19 plot portion of the demo. I won't go through all the code here in this chapter, but you can find the full source code in the plotextdemo in the code folder.

As always, we need to start with our import section. As I have done for the previous projects in this chapter, I will use the `try | except` statements to verify that the user has installed plotext and exit the program if not, but with a quick statement explaining what happened. We also need to include one extra tkinter module, the font module, so we can do some calculations to determine the proper plot size:

```
import sys

try:

    import plotext as plt

except:

    print("This program requires Plotext Library")

    sys.exit()


import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *

import tkinter.font as tkfont

import plotextdemo
```

The next thing that we need to do is to, as always, add a line to the main function to call our startup function and then create that function:

```
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 = plotextdemo.Main(_top1)
    startup()
    root.mainloop()
```

In our startup function, we begin by defining the lists that will provide the options for our TCombobox widgets:

```
def startup():
    markers = [
        "sd",
        "hd",
        "dot",
        "at",
        "heart",
        "smile",
    ]
```

There is one additional marker that Plotext supports and that is the snowflake marker type. However, the "normal" font that we would use for the Label widget does not support that character, at least under Linux, so I have left that out of the option list:

```
    plottypes = [
        "Sine",
        "Scatter",
        "Simple Bar",
```

```
        "Horizontal Bar",
        "Multiple Bar",
        "Log Plot",
        "User Data Plot 1",
    ]
```

Now, we set the values property for the two **Comboboxes** and then create the bindings for them:

```
    _w1.TCombobox1["values"] = markers
    _w1.TCombobox2["values"] = plottypes
    _w1.TCombobox1.bind("<<ComboboxSelected>>", lambda e: on_Marker_
    Select(e))
    _w1.TCombobox2.bind("<<ComboboxSelected>>", lambda e: on_PlotType_
    Select(e))
```

There is one more thing that we need to do for our **Combobox** widgets. That is to set the "default" option for the markers when the program starts up. I decided to start with the "**sd**" marker, which is the option list index **0**. You can choose to use any of the other options; just change the **0** to the index of the marker you want from the list. Remember that in Python, lists are zero-based, so if you choose "**hd**", that would be **1**:

```
    _w1.TCombobox1.current(0)
    global fontname, fontsize, font
    fontname = "DejaVu Sans Mono"
    fontsize = 8
    font = fontname + " " + str(fontsize)
```

The final thing that we need to do is to define the font and font size for the Label that will be used to display our Plots. In my demo, this is _w1.Label3. We need to use a monospaced font (like Courier New or DejaVu Sans Mono) so that each character that makes up the plot will be of the same size as all the others, including the inter-character spacing. I chose a font size of 8 to limit the space that is needed for the display Label. Next is the function **get_plot_size()**. This uses some Tkinter functions to calculate the size, in pixels, and will calculate the proper plotsize values

based on the side of the Label widget. I found that subtracting two pixels from each of the dimensions works best:

```
def get_plot_size():

    global fontname, fontsize, font

    size = _w1.Label3.winfo_width(), _w1.Label3.winfo_height()

    font_plot = tkfont.Font(family=fontname, size=fontsize)

    font_size = font_plot.measure("m"), font_plot.metrics("linespace")
    # in pixels

    size = [size[i] / font_size[i] for i in range(2)]

    size = list(map(int, size))

    cols, rows = size

    return cols - 2, rows - 2
```

Now, we will cover the code for the simplest of the plotext demos, the Sine function plot. First, we clear all the data, then get which of the marker options the user has chosen. Then, we call the `get_plot_size` function to get the proper width and height for the Label and use the `plotsize` method to set that. We then call the `limitsize` method with False, clear the color pallet and have Plotext build the plot, then we use the `.set()` method of the Label to display the plot:

```
def command_plot():

    global selected

    plt.clear_data()

    plt.clear_plot()

    plt.clear_figure()


    selected = _w1.comboMarker.get()

    plt.plot(plt.sin(), marker=selected)


    wid, hei = get_plot_size()

    plt.plotsize(wid, hei)

    plt.limitsize(False)
```

```
    plt.clc()

    canv = plt.build()

    _w1.PlotData.set(canv)
```

I'm going to skip around and describe the two callback functions for the `Comboboxes` next. First, we will examine the `on_Marker_Select()` callback:

```
def on_Marker_Select(e):

    global selected

    selected = _w1.comboMarker.get()

    command_plot()
```

Basically, since this is for the Sine function only, we get the marker type and call the Sine plot function.

The `on_PlotType_select` callback is just as simple; we just get which plot type the user wants, and call that function:

```
def on_PlotType_Select(e):

    global plottype, plottypes

    plottype = _w1.comboPlotType.get()

    if plottype == "Sine":

        _w1.TCombobox1.config(state="normal")

        command_plot()

    elif plottype == "Scatter":

        _w1.TCombobox1.config(state="disabled")

        command_plot_scatter()

    elif plottype == "Simple Bar":

        _w1.TCombobox1.config(state="disabled")

        command_plot_SBar()

    ...

    elif plottype == "User Data Plot 1":

        command_user_data1()
```

Notice that I have removed some of the options in this discussion. Again, you can see all the code in the source code folder. To wrap up, we will look at the user data plot function. Again, due to space, I am going to leave out a large amount of the actual data.

We have already covered the Sine-type plot, now let us look at the Multi-bar plot. This plot is supposed to present the percentages of "approval" for various types of Pizza between Males and Females. The values for men are represented by a more solid bar and for the female, they are represented by bars made up of the dot type markers. The first thing that needs to be done is to set the Combobox for the marker selection to be disabled since we are going to set these in the code for creating the plot. Then, we need to clear the data, plot and figure objects:

```
def command_plot_MBar():

    _w1.TCombobox1.config(state="disabled")

    plt.clear_data()

    plt.clear_plot()

    plt.clear_figure()
```

Since this code came directly from the Plotext website, I cannot attest to the actual percentages used in this demo. However, it does show how to create the various data sets. For this plot, we use three lists to hold the data. The first list holds the different types of pizzas, the second holds the Male "approval" percentages, and the third holds the Female percentages. We load the lists first:

```
    pizzas = ["Sausage", "Pepperoni", "Mushrooms", "Cheese", "Chicken",
    "Beef"]

    male_percentages = [14, 36, 11, 8, 7, 4]

    female_percentages = [12, 20, 35, 15, 2, 1]
```

The next thing to do is to create the plot itself. We call the `multiple_bar` plot function with the X-axis list (which is the pizzas list), then a list holding the Y-axis data (in this case the male and female lists presented as a list of lists), and then the label data for the two types of bars, again as a list, and finally, a list of the markers to use for each Y-axis data set:

```
    plt.multiple_bar(

        pizzas,

        [male_percentages, female_percentages]
```

```
    [male_percentages, female_percentages],
```



```
        label=["men", "women"],
        marker=["sd", "dot"],
    )
```

Finally, we set the title of the plot, provide the **plotsize** information, set the **limitsize** method to False, clear the colors, build the plot, and set it into the Label via the **.set()** method of our display label:

```
    plt.title("Most Favored Pizzas in the World by Gender")
    # plt.plotsize(150, 35)
    wid, hei = get_plot_size()
    plt.plotsize(wid, hei)
    plt.limitsize(False)
    plt.clc()
    canv = ""
    canv = plt.build()
    _w1.PlotData.set(canv)
```

That is it. It really cannot get much easier than that, can it?

To wrap up, we will look at the user data plot function. We do this the same way we have done all the other plots, clear the objects, set up the data, assign the data to the specific plot type we need for the X-axis and Y-axis data, the labels for the plot, and then build the plot and place it into the Label for display:

```
def command_user_data1():
    _w1.TCombobox1.config(state="disabled")
    plt.clear_data()
    plt.clear_plot()
    plt.clear_figure()
```

The data points for this plot are from a spreadsheet that was put out on weekdays during the crisis. We simply pulled the data for a series of days and the days of the year directly from the spreadsheet into two lists:

```
    # ================================================
```

```
# Covid-19 data from State of Texas Published Spreadsheet
# ================================================
data = [
    21146,
    21183,
    21197,
    ...
    22305,
    22420,
]
datedata = [
    "6/27/21",
    "6/28/21",
    "6/29/21",
    ...
    "7/26/21",
    "7/27/21",
]
```

Because this is a "**date type**" plot, we need to set up things in a bit different order and into different methods. First, we need to set the form that the date will be displayed. Here, we have used the date format of "**month/day/year**". Then, we provide the `plot_date` method with the `datedata` (y-axis data), then the `data` (X-axis data) lists, the title for the plot, and the y axis label (`Confirmed Cases`). We set the `plotsize` information, the `limitsize` boolean, again clear the color information, build the plot and send it to the Label widget for display:

```
plt.datetime.set_datetime_form(date_form="%m/%d/%y")
plt.plot_date(datedata, data)
```

```
plt.title("Covid-19 Hays County, Tx. Data from 06/27/2021 -
07/27/2021")
```



```
plt.ylabel("Confirmed Cases")

# plt.plotsize(150, 35)

wid, hei = get_plot_size()

plt.plotsize(wid, hei)

plt.limitsize(False)

plt.clc()

canv = ""

canv = plt.build()

_w1.PlotData.set(canv)
```

Please note that in all the Plotext demos, I have hard-coded the lists that provide the data for the plots. The data could have been pulled directly from the database or spreadsheet through many of the data science packages like Numpy or Pandas into the lists via code. That, however, in the case of our demos for the book was unrealistic.

Conclusion

Now, you learned how to use the Custom Widget holder and how to call third-party libraries from your PAGE projects. For the most part, most third-party widgets and libraries are just as easy as the ones that are shown. The biggest challenge for you will be understanding the initialization routines that those libraries require.

One of the third-party libraries for extra widgets is the ttkwidgets package, which you can find at **https://github.com/TkinterEP/ttkwidgets**. These extra widgets were designed for use when programming a project directly from Tkinter source code and when using PAGE, many of the widgets are already in the PAGE toolbox. However, there are many other things that this package brings that can help you, like the color picker and the font picker.

Another valuable third-party widget is the **tksheet** spreadsheet widget which you can find at **https://github.com/ragardner/tksheet**. Be sure to read the documentation to use this library.

In the next chapter, we will look at using PAGE to be a frontend for SQL database

programs.

Multiple Choice Questions

1. What is the easiest way to get extra widgets for use with PAGE?
   a. Write your own.
   b. You can't!
   c. Search the Internet.

Answer: The answer is a combination of **c) Search the Internet** and **a) Write your own**. Always start with a search of the Internet first, since many widgets have already been written for you.

2. Can I add Custom widgets to the PAGE Toolbox myself?
   a. YES
   b. NO
   c. You can but it is very hard.
3. Will any third-party Custom widget work with PAGE?
   a. YES
   b. NO
   c. It depends.

Answers

1. The answer is a combination of **c) Search the Internet** and **a) Write your own**. Always start with a search of the Internet first, since many widgets have already been written for you.
2. B – No. PAGE is written in Visual TCL and there is no option to directly add a Custom Widget to the PAGE toolbox.
3. C – It depends. For the most part, you SHOULD be able to incorporate most any third-party custom widget that is written for Tkinter. However, some take so much initialization code, it might be easier to find another option. You can always ask for help from the author of the widget or the PAGE discussion form or the PAGE Discord channel.

Questions

1. What is the best way to include a Custom Widget holder in your PAGE project?
2. Do I need to use the Custom Widget holder anytime I want to include a third-party Tk or ttk widget?
3. How do I know how what size to make the Custom Widget holder in the form itself?

Answers to the questions

1. Place a Frame on your Toplevel form and set the size to what you think your widget will need. Then, drop the Custom Widget holder into the frame, use mouse button 3 (right click) on the Custom Widget holder, and select Widget | Fill Container to size the widget to fill the Frame.
2. That depends on the widget. The TkDatePicker widget has two ways to use it. One requires the use of the Custom Widget holder, but the other allows you to call it directly from the internal Application Programming Interface (API).
3. You need to guess. Most of the widgets provide some sort of demo that will give you a good clue as to the size that would be taken up by the widget. And then since you get the code for the demo, all you have to do is look at the code to determine the proper size. Also, remember that things like font size will cause the widget size to change. Because you (hopefully) placed the Custom Widget holder inside of a Frame and had the Custom widget holder fill that frame, it is easy to move and resize the Frame, thus moving and resizing the Custom Widget at the same time to wherever and whatever size you might need.

Key terms

- Third-party Widgets
- Custom Widget
- API
- Date Picker
- Color Picker
- ScrolledCheckedbuttonListbox

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Creating a SQLite Database Front End

Introduction

Data flows at a rate that was never seen before and will only increase. However, knowing what data to use and where to get it is the real trick.

In this chapter, we will create a front end to a database using PAGE. When I say "front end", it simply means that the PAGE program will allow you to display, edit, and add data from the database itself.

There are many types of databases in the world; some are located on servers and shat are local. In our case, the database is an SQLite database that is local to your hard drive. SQLite uses a subset of the **Structured Query Language (SQL)** to query or request information from the database itself, add new records to the database,

and edit existing information. The SQLite engine is part of Python, so we do not need anything special to access the data.

We will be doing things a bit differently in this chapter. You already know how to create a GUI in PAGE, so I will show you images of the GUI and I will point out some of the new or different things that I used to create the demo form. Then, we will concentrate on the code where the magic happens. Before we can get to our form and code, however, we need to create the database and know what data we might deal with.

What you will learn

We will learn the following topics in this chapter:

- General Database information
- SQLite Database information
- Displaying Database Information
- Handling Database Entry Information
- Using a theme in your GUI
- Dynamically switching Themes
- Creating a ttk Toolbar Menu
- Creating a Navigation bar
- Displaying the Current Time in your GUI
- Using the TOML format configuration files

Things you will need

We will use some third-party tools and Python libraries to make our application, but all of them are free and available for all operating systems.

First, the Python libraries. We will need to install the extra libraries using pip or pip3 depending on your Python installation. The first two are named **`tomli_w`** and **`tomli`** which are used to create the configuration files in the **TOML** format. Install them using **pip** or **pip3**:

```
pip install tomli

pip install tomli_w
```

Another third-party library is one that I wrote to help me with using Database files

called **dbutils.py**. You do not have to install this one. Just make sure it is your project folder with your Python code. I have included it in the code folder.

Finally, we will use very few global variables in our program; instead, we will use the **shared file scheme**. We have discussed this in earlier chapters. All this requires having an empty file in the same folder as your code named **shared.py**.

As to tools, you do not actually absolutely need to have these, but they will make your life much easier when dealing with SQLite files. The first is named **DB Browser for SQLite**. It is free and available for all operating systems. You can download it from their website at **https://sqlitebrowser.org/**.

The second tool is not absolutely needed but is nice to have. It is called **SQLeo** and again is available for all operating systems. **SQLeo** is a visual query design tool that allows you to create your queries much easier. You can get it from **https://sqleo.sourceforge.io/**. The installation is a bit difficult, so be sure to read the instructions. It requires Java so be sure to install that first. You will also need a driver to support SQLite databases. You can read how to download it and install the driver at **http://kenfallon.com/adding-sqlite-as-a-datasource-to-sqleo/**.

Getting started – SQLite databases at a glance

Before we get too deep into trying to design a set of forms to work with databases, we need to understand what a database really is and get some of the concepts under our belts.

Think of a spreadsheet. You can have a spreadsheet with multiple pages. Each page is based on a subject. Each page is broken into rows and columns. The rows, just like our computer screen go from the left to the right and the columns go from the top down. Where a row and column meet, that is called a cell. The cell can hold data about that particular page.

A database can have a group of data called a table, and the database can have multiple tables. In each table, you have what are called fields which describe some data that pertains to the table. So, the table is like the spreadsheet page and the fields are like the spreadsheet columns. Database tables also have rows that hold information about each column or field.

information about each column or field.

So, if we had a database that was about recipes, we can visualize it as shown in *Figure 9.1*:

	idRecipes	RecipeText	RecipeSource	RecipeServes	TotalTime
	Filter	Filter	Filter	Filter	Filter
10	19	Nicole's Wonderful White Chili	web	4	NULL
11	20	Texas White Chili	web	4	NULL
12	21	Sloppy Joe Seasoning Mix	Web	6	NULL
13	22	Snow Balls in a Jar	web	4 dozen	NULL
14	23	BBQ Cowboy Beans	web	6	NULL
15	24	Lemon Supreme Pie	Web	6-8	NULL
16	25	Pineapple Chiffon Pie	web	6-8	NULL
17	26	Pumpkin Cookies	Carol		NULL
18	27	Easy Creamy Key Lime Pie	www.realemon.ca/en/recipes/...	6-8	None
19	31	Chimichuri Sauce for Steaks	Foodnetwork.com	1 1/2 cups	NULL
20	32	Boudin Sausage	Emeril Live	4 1/2 pounds	NULL
21	34	Chipotle Cashew Chicken with Brown Rice 2	Rachael Ray	4-6	NULL
22	35	Crispy Chicken Cutlets with Basil-Parsley Sauce	Rachael Ray	4	NULL
23	36	Cheesy Risi e Bisi	Rachael Ray	4	NULL
24	37	Basil- Parsley Sauce	Rachael Ray	4	NULL
25	38	Strip Steak with Pepper Cream Sauce	Alton Brown	4	NULL
26	39	Sweet and Sour Pork	Alton Brown	4-6	NULL

Figure 9.1: Database table

When you look at it like this, it does resemble a spreadsheet page, doesn't it? In this particular table, there is a record number, the name of the recipe, where the recipe came from, how many servings it makes and the total time, and a few others. But what it does not have is any fields for the ingredients that make up the recipe or any instructions. What good is a list of recipes that only has a name of the recipe and no ingredients or instructions?

This database also includes a table that has all the ingredients (the amount of each ingredient) and a separate table for instructions.

Creating the project database

You can use the DB Browser for SQLite tool to create your entire database structure and even add the information for the various records for the tables. However, I have created a simple Python file that will do all of this for you. Let us look at the code for this.

Of course, we need to start with the import section:

```
import sqlite3

import sys

import os
```

Then, we need to create our main function that will directly handle all the **worker** functions:

```
def startup():

    global connection, cursor, database_name

    database_name = "Company.db"

    open_database()

    make_employee()

    make_company()

    make_departments()

    fill_company()

    fill_employees()

    fill_departments()
```

You can see that there is very little in the main function. As I said, it just calls the various functions in the correct order.

The first thing we must do is define the name of the database. One thing you MUST remember about using SQLite databases. When you try to open the database, if it does not exist, a blank database will be created. That is good if you are creating the database from code, but if you are trying to use a created database, you must get the name and location of the database file correct:

```
def open_database():

    global connection, cursor

    dbpath = os.path.join(location, database_name)

    connection = sqlite3.Connection(dbpath)

    cursor = connection.cursor()
```

We are using two global variables here, **connection** and **cursor**. Those were defined in the startup function and are duplicated here. We simply use the location of the project and add the name of the database ("**Company.db**"), then we use the connection objct to connect, or open the database file. At this point, it does not exist so SQLite will create it. Then, we create a cursor object. The cursor object does most of the work talking to the database file itself.

Once we have the database created and the connection and cursor objects created, we can start defining our database tables. The tables must be created before we can add any information to them:

```
def make_employee():
    sql1 = """CREATE TABLE "Employees" (
   "EmpID"   INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT UNIQUE,
   "FirstName"  TEXT,
   "MiddleName" TEXT,
   "LastName"   TEXT,
   "Address1"   TEXT,
   "Address2"   TEXT,
   "City" TEXT,
   "State"   TEXT,
   "PostalCode" TEXT,
   "DateOfBirth"TEXT,
   "PhoneNumber"TEXT,
   "DepartmentID"   INTEGER,
    "Active"        INTEGER
)"""
    cursor.execute(sql1)
```

The `make_employee`Notice that most of the fields are text based. The three exceptions are the **primary key** for the record, the `DepartmentID`oolean value (0 or 1). When we 'delete' an employee from the database, we simply set this flag to 0. That way, if we need to find it later, it is still there. When we want to see all the current employees, we simply need to look for those records that have a 1 in the **Active** field. We end the function with the command to execute the SQL statement that we created. This is a job that the cursor object does.

There will be two other create table functions, that we will not discuss, since they are essentially the same as the `make_employee` function, just with different fields in a different table.

Next, we will fill in the base data for the three tables. Again, we will only look at filling in the Employee data table:

```
def fill_employees():
    data = [
        [
            "Estelle",
            "Lamar",
            "Vipond",
            "1 Coach House Rd",
            "",
            "Austin",
            "Texas",
            "78745",
            "09/10/1958",
            "512-555-0291",
            6,
            1,
        ],
        ...
    ]
```

You can see that we are creating a list of lists named data. Each list within the data list contains one record about a single employee. It contains one entry for each field in the employee data table except for the **Primary key** field (EmpID). That is because we informed the SQLite engine that this field is supposed to be **auto-incremented** to guarantee each record has a unique number:

```
    for dat in data:
        print(len(dat))
```

Now, we walk through the data, one record at a time, creating an SQL statement to insert the data into the record. We use a safe SQL statement and load the data into the spaces where the question marks are in the string:

```
        sql1 = """INSERT INTO "Employees" (FirstName,MiddleName,LastNam-
        e,Address1,Address2,City,State,PostalCode,DateOfBirth,PhoneNum-
        ber,DepartmentID,Active) VALUES (?,?,?,?,?,?,?,?,?,?,?,?);"""

        info = (

            dat[0],

            dat[1],

            dat[2],

            dat[3],

            dat[4],

            dat[5],

            dat[6],

            dat[7],

            dat[8],

            dat[9],

            dat[10],

            dat[11],

        )

        cursor.execute(sql1, info)

    connection.commit()
```

Finally, we have the cursor object execute the SQL statement, and then use the commit function in the connection object to write the data.

The GUI form

In this chapter, I am not going to give you instructions on creating the GUI. Instead,

I am going to show you the GUI and important portions of it and allow you to create it yourself. The actual GUI file is available in the code folder for *Chapter 9, Creating a SQLite Database Front End* .

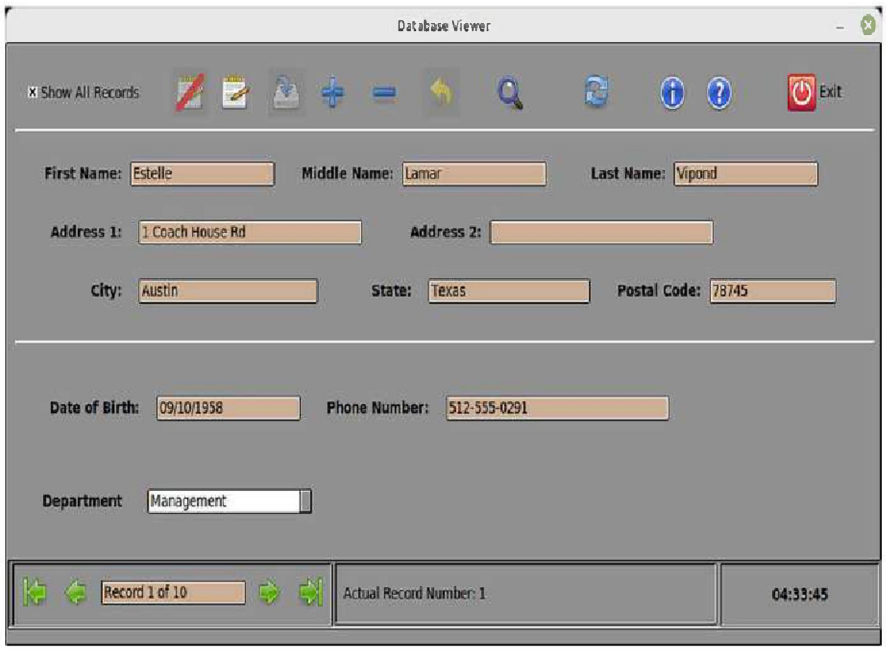


Figure 9.2: Database Viewer GUI

In this project, all the widgets are ttk widgets. We will use (as shown in the following screenshot) a theme that I created called **notsodark**. You can see at just a glance, we have many TButtons, Tentry widgets, Tframes, a Tcombobox, and a Tcheckbutton. The first widget on the form is a Tframe that has been expanded to fill the entire form and has been locked so it cannot be accidentally moved. All other widgets are placed on top of the frame:

Figure 9.3: Toolbar

The Toolbar is not in a separate frame, just placed all in a line. There is a Tseparator that separates the buttons from the rest of the form. All the Tbuttons have the style attribute set to Toolbutton, which will make them appear flat until the mouse cursor hovers over them as shown in *Figure 9.4*:

Figure 9.4: Navigation bar

The navigation bar is placed in a Tframe first, then created in another frame. There are four Tbuttons (all with the style set to Toolbutton) with a Tentry widget in the

middle. This bar allows the user to display the first, previous, next, and last records in the database. The Tentry displays the current record and the total number of records for that query:

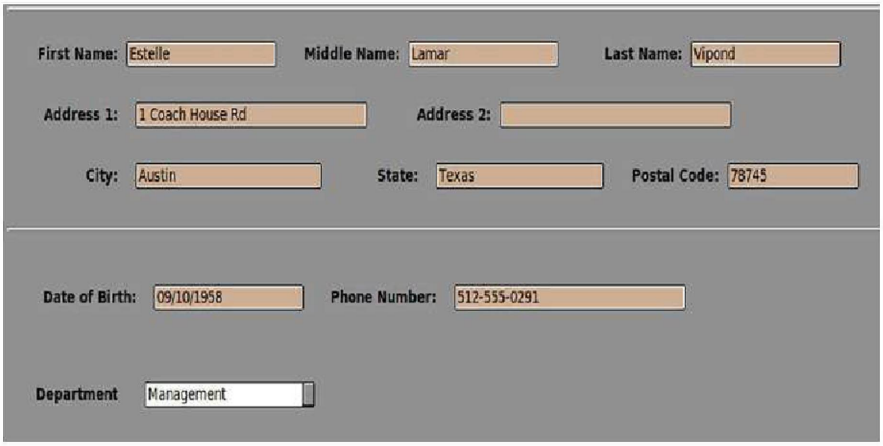


Figure 9.5: Entry Section

The entry section of the form contains Tlabels and Tentry widgets, a Tseparator in the middle and a Tcombobox at the bottom to allow the display and selection of the department that the employee is in.

Now, we will look at writing the code.

Writing the Code

The entire code for the project will not be given in this chapter text, but the entire project (as always) is available in the *Chapter 9, Creating a SQLite Database Front End* code folder. I will show you the majority of the code, since it is important for you to understand the hows and whys of what we are doing.

First, the import section. We need to have a number of supporting modules for this project. Be sure to install the TOML files as I instructed earlier:

```
import sys

import os

import platform

import datetime

import sqlite3

import dbutils
```

```
from dbutils import quote

import tomli_w

import tomli

import shared
```

Remember the shared module is just an empty file named shared.py so we do not have to have so many global variables.

We also need a couple of extra imports from tkinter. These should go after the normal tkinter imports:

```
import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *

from tkinter import messagebox

from tkinter import simpledialog
```

We also will retrieve the physical location of the code in the file system and name it location. This comes from the GUI file. We use it to locate the database:

```
location = dbview._location
```

As usual, we need to modify the main function to add a call to the startup function. It is placed in the normal place.

The startup function has a lot to do so it is fairly long. I have added a lot of comments in the code to help you along:

```
def startup():

    shared.showDeleted = False

    # Get startup information from the configuration file

    shared.config_data = get_configuration_information()

    showdeletedrecords = shared.config_data["Startup"]["showdeleted"]

    if showdeletedrecords == True:

        _w1.tch47.set(1)

        shared.showDeleted = True
```

```
    else:

        _w1.tch47.set(0)

        shared.showDeleted = False
```

As I said earlier, when records are deleted, we really do not delete them, we just mark them as not active. So, we need to start off with the `showDeleted` flag as False. We then read the configuration file and set the `showDeleted` flag to what the configuration says. We then set the TCheckedbutton to properly reflect the value of the flag. Next, we show, in the terminal, the version of Python that is being used to run the program and some other information. Then, we set a variable to the `ttk.Style` object so we can control the themes that will be used to control the look of the program. Then, we load the themes into memory through the `load_tcl_themes` function and set the startup theme to what the configuration file says. The `TLabel` widgets do not respond to themes correctly, so we call a function called `fix_labels` to set the background properly:

```
    show_program_info()

    sty = ttk.Style()

    # Load the various theme .tcl files

    load_tcl_themes()

    # set the theme

    theme = shared.config_data["Theme"]["current"]

    sty.theme_use(theme)

    # TLabels don't properly respond to Themes.  Fix the Label backgrounds

    fix_labels()
```

At this point, we start loading the database information. We set the name of the database, open it (setting the connection and cursor objects), and start gathering the data. First, we load the department information in the `load_departments` function and fill the `TCombobox` with it. Then, we query the database to get all the employee records in the `get_emps` function and finally load the form with the first employee record in the database through the `load_form` function:

```
    database_name = "Company.db"

    open_database(database_name)

    # Get the department information from the database
```

```
depts = load_departments()

# Load that information into the TCombobox

_w1.TCombobox1["values"] = depts

# Get the employee records from the database

shared.actual_records = 0

get_emps()

global emprecs

shared.num_records = len(emprecs)

shared.currentrec = 1

# Load the form with the first record

load_form("id", shared.currentrec)
```

The next thing we do is set up the routines to show the time of day in the bottom right label of the status bar. We then disable the *Restore* button, the *unEdit* button, and the Entry widget in the *navigation bar*, so the user cannot accidentally change them. Next, we set all of the `TEntry` widgets to disabled (since we are starting in the viewer mode) using the `setup_for_edit` function. Finally, we set the icon for the program and center the form on the screen:

```
shared.timer_id = root.after(0, on_time_update)

shared.edit_mode = ""

_w1.btnRestore.configure(state=DISABLED)

_w1.btnUnEdit.configure(state=DISABLED)

_w1.TEntry1.configure(state=DISABLED)

# Start in the display mode..."Lock" all the TEntry widgets

setup_for_edit(True)

# Set the icon for when the app is minimized

set_icon()

# Finally, center the form in the monitor screen

centre_screen(998, 516)
```

Now, we need to deal with all the support functions that have been called within the startup function and those that will be used during the run of the program.

First, we will do the `setup_for_edit` function. This will disable or enable all the Entry widgets. We disable them for the viewer mode and enable them for the edit or new entry mode. To do this, we create a list of all the widgets we need to deal with, the walk through the list and set them properly:

```
def setup_for_edit(lock=True):
    """This function will "lock" the TEntry widgets if lock=True,
    so that no changes can be made by accident."""
    widgetlist = [
        _w1.TEntFName,
        _w1.TEntMName,
        _w1.TEntLName,
        _w1.TEntAddr1,
        _w1.TEntAddr2,
        _w1.TEntCity,
        _w1.TEntState,
        _w1.TEntPostal,
        _w1.TEntDOB,
        _w1.TEntPhone,
        _w1.TCombobox1,
    ]
    for widget in widgetlist:
        if lock:
            widget.configure(state=DISABLED)
        else:
            widget.configure(state=NORMAL)
```

Now, we need to create the functions that read and write the TOML configuration file. It is a very simple human readable file that holds information about how the application will work at startup which is read at startup by the `get_configuration_information` function. This includes the theme that we will use, the theme to use at startup and we show the "deleted" records at startup. If the user changes the theme during the run of the program or changes the show all records feature, these changes will immediately be set into the configuration file by the `update_config_data` function:

```
def get_configuration_information():
    """This function reads the TOML configuration file into a dictionary
    for use as needed."""

    with open("config.toml", "rb") as f:
        shared.config_data = tomli.load(f)
    return shared.config_data


def update_config_data():
    """This function writes the TOML configuration file, usually is called
    when
    a change has been made to the configuration dictionary"""
    with open("config.toml", "wb") as f:
        tomli_w.dump(shared.config_data, f)
```

Next, we have the function that loads the theme files so we can read them. Remember that these are Tcl files, so we must load them in a special way:

```
def load_tcl_themes():
    _top1.tk.call("source", os.path.join(location, "themes",
    "notsodark_0.1.05.1.tcl"))
    _top1.tk.call("source", os.path.join(location, "themes", "waldorf.
    tcl"))
```

The `load_form` function is used every time a record is shown. We get one record from the employee records we have already gathered in the startup function. We need to pass if we will be using the actual record number or the record ID as well

as the record number we want to show. The actual record number is a special field that was added to the end of the Recordset when we ran the query to get the records:

```
def load_form(how, which):
    global emprecs, temprec
    clear_form()
    shared.num_records = len(emprecs)
    if how == "id":
        if which > shared.num_records:
            which = shared.num_records
        recnum = which
        disp = f"Record {recnum} of {shared.num_records}"
        _w1.EntryRecordInfo.set(disp)
        if how == "actual":
            pass
        rec = emprecs[which - 1]
    elif how == "actual":
        rec = emprecs[which - 1]
```

Since we already have the records into a list of lists, we do not need to requery the **Employee** datatable. All we need to do is get the list that contains the records we want and enter them into the Entry widgets. When we go to display the department that employee works in, we carried the number of the **Department** record in the employee Recordset, so we just need to set the TCombobox to show the proper record in it is list of departments. Finally, if the record has the **Active** flag to 0, we add the word "**deleted**" to the actual record display. If not, we do not show anything at the end:

```
    shared.current_record_data = rec
    _w1.EntryFirstName.set(rec[1])
    _w1.EntryMiddleName.set(rec[2])
    _w1.EntryLastName.set(rec[3])
```

```
    _w1.EntryAddr1.set(rec[4])
    _w1.EntryAddr2.set(rec[5])
    _w1.EntryCity.set(rec[6])
    _w1.EntryState.set(rec[7])
    _w1.EntryPostal.set(rec[8])
    _w1.EntryDOB.set(rec[9])
    _w1.EntryPhone.set(rec[10])
    _w1.TCombobox1.current(rec[11] - 1)
    if rec[12] == 0:
        dele = " deleted"
        _w1.btnRestore.configure(state=NORMAL)
        _w1.TButton9.configure(state=DISABLED)
    else:
        dele = " "
        _w1.btnRestore.configure(state=DISABLED)
        _w1.TButton9.configure(state=NORMAL)
    _w1.RecordNumber.set(f"  Actual Record Number: {rec[0]} {dele}")
```

We have come to the actual function that reads the **employee** data table. We need to use SQL statements to do this. The first SQL statement gets all the records from the **employee** table, regardless of the Active flag as well as adds a field, called **RecNumber** that holds the count of the record as it is being added to the recordset. The second SQL statement only retrieves the records that have a 1 in the **Active** field and adds the count of the records in the recordset into the **RecNumber** field. We use this special field to navigate through the first, next, previous, and last commands. If we cannot get the employee records, we notify the user and end the program:

```
def get_emps():
    global emprecs
    if shared.showDeleted:
        sql = """SELECT *, COUNT(*) OVER(ORDER BY EmpID) as RecNumber
```

```
FROM Employees"""
    else:
        sql = """SELECT *, COUNT(*) OVER(ORDER BY EmpID) as RecNumber
FROM Employees where Active = 1"""
    emprecs = list(cursor.execute(sql))
    if emprecs:
        size = len(emprecs)
        actual_records = emprecs[size - 1][13]
        shared.actual_records = actual_records
    else:
        titl = "Database Viewer"
        msg = "Can't get Employee Records from Database.  Program will
        exit!"
        messagebox(titl, msg, parent=_top1, icon=messagebox.error)
        sys.exit()
```

The `open_database` function creates the connection and cursor objects that allows us to do all our database queries:

```
def open_database(database_name):
    global connection, cursor
    dbpath = os.path.join(location, database_name)
    connection = sqlite3.Connection(dbpath)
    cursor = connection.cursor()
```

The `load_departments` function does one of the simplest SQL statements to get all the department names from the `Department table`, loads it into a list, and returns it. If, for some reason, it cannot get the data, it warns the user and ends the program:

```
def load_departments():
    sql = "SELECT DeptName FROM Department;"
    recs = list(cursor.execute(sql))
```

```
    lst = []
    for r in recs:
        lst.append(r[0])
    if len(recs) < 1:
        titl = "Database Viewer"
        msg = (
            "No records were returned for the Department table.  Program
            will now exit."
        )
        messagebox.showerror(titl, msg, parent=_top1, icon=messagebox.
        ERROR)
        sys.exit()
    return lst
```

The next four functions control the navigation through the various Employee records. The name of the function describes what the function does. `on_btnGoFirst` jumps to the first record of the recordset, `on_btnGoLast` jumps to the last record. `on_btnGoNext` moves to the next record in the recordset, and `on_btnGoPrevious` goes to the previous record. They all call the `load_form` function to show the record on the form:

```
def on_btnGoFirst(*args):
    """Callback for the goFirst button"""
    shared.currentrec = 1
    load_form("id", shared.currentrec)


def on_btnGoLast(*args):
    shared.currentrec = shared.num_records
    load_form("id", shared.currentrec)


def on_btnGoNext(*args):
    if shared.currentrec < shared.num_records:
```

```
        shared.currentrec += 1
        load_form("id", shared.currentrec)
    else:
        pass

def on_btnGoPrevious(*args):
    if shared.currentrec > 1:
        shared.currentrec -= 1
        load_form("id", shared.currentrec)
```

The `on_btnChangeTheme` function switches between the two themes then calls the `theme_use()` method from the `ttk.Style` object to cause the form to use the proper theme. Finally, it calls the `fix_labels` function.

```
def on_btnChangeTheme(*args):
    current = shared.config_data["Theme"]["current"]
    if current == "notsodark":
        current = "waldorf"
    else:
        current = "notsodark"
    shared.config_data["Theme"]["current"] = current
    sty = ttk.Style()
    sty.theme_use(current)
    fix_labels()
```

Since the `ttk::TLabel` does not respond properly to the `TLabel` style (if any) in a ttk Theme, the `fix_labels` function gets the background color from the TFrame that is the container for all the widgets in our form by using the lookup function from the ttk Style object. Then, we walk through a list of all the TLabel widgets and sets the proper background one by one:

```
def fix_labels():
```

```
    sty = ttk.Style()
    _bg = sty.lookup("TFrame", "background")
    update_config_data()
    labels = [
        _w1.TLabel1,
        _w1.TLabel2,
        _w1.TLabel3,
        _w1.TLabel4,
        _w1.TLabel5,
        _w1.TLabel6,
        _w1.TLabel7,
        _w1.TLabel8,
        _w1.TLabel9,
        _w1.TLabel10,
        _w1.TLabel11,
        _w1.TLabel12,
        _w1.TLabel13,
    ]
    for lab in labels:
        lab.configure(background=_bg)
```

The `find_by_name` function simply calls the `simpledialog.askstring` function and returns whatever the user entered:

```
def find_by_name():
    resp = simpledialog.askstring(
        "Search for Last Name", "Enter Last Name to search for:"
    )
    return resp
```

The **on_btnSearch** function starts off by calling the **find_by_name** function (above), then creates a simple SQL statement that searches the **LastName** field of the Employee table using the like query. That way, if we want to find "Kirk", the SQLite engine will match **Kirk**, **kirk**, or even **kir**. If the statement finds a match, the program clears the form, pulls the correct record, fills the form with the record, and displays a success message. If not, a failure message is displayed:

```
def on_btnSearch(*args):

    resp = find_by_name()

    if resp != None:

       sql = f"""SELECT * FROM Employees WHERE LastName like '%{resp}%'"""

        found = list(cursor.execute(sql))

        if found:

            rec = int(found[0][0])

            clear_form()

            load_form("id", rec)  # Records are a list which is 0 based

            titl = "Database Viewer"

            msg = f"Search Success! Found {resp}"

            messagebox.showinfo(titl, msg, parent=_top1,
            icon=messagebox.INFO)

        else:

            titl = "Database Viewer"

            msg = f"'{resp}' not found in database!"

            messagebox.showerror(titl, msg, parent=_top1,
            icon=messagebox.ERROR)
```

The `clear_form` function is another simple function that walks through a list of widgets and uses the `.set()` method to clear whatever is being displayed in it:

```
def clear_form():

   widgetlist = [

        _w1.EntryFirstName,
```

```
        _w1.EntryMiddleName,
        _w1.EntryLastName,
        _w1.EntryAddr1,
        _w1.EntryAddr2,
        _w1.EntryCity,
        _w1.EntryState,
        _w1.EntryPostal,
        _w1.EntryDOB,
        _w1.EntryPhone,
    ]
    for widget in widgetlist:
        widget.set("")
    _w1.TCombobox1.set("")
```

The `on_btnAdd` function sets a shared variable named `edit_mode` to `"new"`, clears the form, sets the focus to the `TEntFName` Entry widget, changes the state of the `btnSave` widget from *disabled* to *normal*, and calls the `setup_for_edit` function to unlock all the widgets that will be used:

```
def on_btnAdd(*args):
    shared.edit_mode = "new"
    clear_form()
    _w1.TEntFName.focus_set()
    _w1.btnSave.configure(state=NORMAL)
    setup_for_edit(False)
```

The next two functions will disable or enable the navigation movement buttons. This is so when in either the *edit* or *new* mode, the movement buttons do not work:

```
def disable_record_move():
    _w1.TButton1.configure(state=DISABLED)
    _w1.TButton2.configure(state=DISABLED)
```

```
    _w1.TButton3.configure(state=DISABLED)

    _w1.TButton4.configure(state=DISABLED)


def enable_record_move():

    _w1.TButton1.configure(state=NORMAL)

    _w1.TButton2.configure(state=NORMAL)

    _w1.TButton3.configure(state=NORMAL)

    _w1.TButton4.configure(state=NORMAL)
```

The `on_btnDelete` function changes the `Active` field for the current record from 1 to 0 after prompting the user to make sure they want to delete it:

```
def on_btnDelete(*args):

    titl = "Database Viewer"

    msg = "This will remove the current record from the Database.
    Proceed?"

    resp = messagebox.askyesno(titl, msg, parent=_top1, icon=messagebox.
    QUESTION)

    if resp == True:

        sql = f"""UPDATE Employees SET Active=0 WHERE EmpID={shared.
        currentrec}"""

        cursor.execute(sql)

        connection.commit()

        get_emps()

        load_form("id", shared.currentrec)
```

The `on_btnEdit` sets the shared `edit_mode` to "`edit`", and then sets the proper states of various buttons for the edit mode, and checks whether the current record is Active or not. Basically, it takes care of the housekeeping for the mode:

```
def on_btnEdit(*args):

    temp_data = shared.current_record_data

    shared.edit_mode = "edit"
```

```
    _w1.btnUnEdit.config(state=NORMAL)

    _w1.btnEdit.config(state=DISABLED)

    disable_record_move()

    if temp_data[12] == 0:

        # record is deleted.  Complain!

        titl = "Database Viewer"

        msg = "The current record is marked as Deleted.  Restore the
        record First!"

        messagebox.showerror(titl, msg, parent=_top1, icon=messagebox.
        ERROR)

    else:

        _w1.btnSave.configure(state=NORMAL)

        # May not be needed

        current_record = shared.currentrec


        _w1.btnSave.configure(state=NORMAL)

        setup_for_edit(False)
```

The **on_time_update** function is the callback for the **root.after** Tk timer function. It gets called when the timer expires, does what it needs to (in this case, shows the current time in the time label), and then sets the next timer length and goes back to sleep:

```
def on_time_update():

    nowstring = f"{datetime.datetime.now():%X}"

    _w1.StatusTime.set(nowstring)

    shared.timer_id = root.after(500, on_time_update)
```

This is the callback for the **on_btnSave** button. It does not actually do the saving, but it does check all the fields to see which fields have been changed if we are in the **"edit"** mode or causes the new record to be written to the database if we are in the **"new"** mode. The two modes need to be handled differently. If we are in the **"edit"** mode, we need to use the **update SQL** statement since we will be updating

the existing record. If we are in the "new" mode, we just need to use the **insert SQL** statement, since we will be creating a new record:

```
def on_btnSave(*args):
    shared.fields = []
    if shared.edit_mode == "edit":
        shared.fields = [
            "FirstName",
            "MiddleName",
            "LastName",
            "Address1",
            "Address2",
            "City",
            "State",
            "PostalCode",
            "DateOfBirth",
            "PhoneNumber",
            "DepartmentID",
            "Active",
        ]
        check_for_changes(shared.currentrec)
    elif shared.edit_mode == "new":
        write_full_record()
        setup_for_edit(True)
        enable_record_move()
    else:
        pass
```

The `check_for_changes` function compares the data in the Entry widget to the proper field in the recordset. If they are different, then we set the field number into a list and the changed data into another list. Once all the fields have been compared and if there have been any changes, the `update_record` function is called:

```
def check_for_changes(recnum):
    original_data = list(shared.current_record_data)
    changedfields = []
    newdata = []
    if _w1.EntryFirstName.get() != original_data[1]:
        changedfields.append(0)
        newdata.append(_w1.EntryFirstName.get())
    if _w1.EntryMiddleName.get() != original_data[2]:
        changedfields.append(1)
        newdata.append(_w1.EntryMiddleName.get())
    if _w1.EntryLastName.get() != original_data[3]:
        changedfields.append(2)
        newdata.append(_w1.EntryLastName.get())
    if _w1.EntryAddr1.get() != original_data[4]:
        changedfields.append(3)
        newdata.append(_w1.EntryAddr1.get())
    if _w1.EntryAddr2.get() != original_data[5]:
        changedfields.append(4)
        newdata.append(_w1.EntryAddr2.get())
    if _w1.EntryCity.get() != original_data[6]:
        changedfields.append(5)
        newdata.append(_w1.EntryCity.get())
    if _w1.EntryState.get() != original_data[7]:
        changedfields.append(6)
```

```
        newdata.append(_w1.EntryState.get())
    if _w1.EntryPostal.get() != original_data[8]:
        changedfields.append(7)
        newdata.append(_w1.EntryPostal.get())
    if _w1.EntryDOB.get() != original_data[9]:
        changedfields.append(8)
        newdata.append(_w1.EntryDOB.get())
    if _w1.EntryPhone.get() != original_data[10]:
        changedfields.append(9)
        newdata.append(_w1.EntryPhone.get())
    # Department needs to come from the Combobox
    dept = quote(_w1.combobox.get())
    dat = get_department_id(dept)
    if dat != original_data[11]:
        changedfields.append(10)
        newdata.append(dat)
    if _debug:
        print(f"Changed fields: {changedfields}")
        print(f"New Data: {newdata}")
    update_record(recnum, changedfields, newdata)
```

The `update_record` function will create an updated SQL statement on the fly using the changed field information. Once it is finished creating the SQL statement, it then calls the cursor object to execute the statement and finally calls the `connection.commit` to force the database engine to write the data:

```
def update_record(which, changedfields, newdata):
    sql1 = f"UPDATE Employees SET"
    endsql = f" WHERE EmpID = {which}"
    cntr = 0
```

```
numtodo = len(changedfields)

parts = []

for field in changedfields:

    print(f"{field=} - {shared.fields[field]=} - {newdata[cntr]=}")

    if field == 10:

        sqltmp = " " + shared.fields[field] + f" = {newdata[cntr]}"

    else:

      sqltmp = " " + shared.fields[field] + " = " + quote(newdata[cntr])

    parts.append(sqltmp)

    cntr += 1

sql = ",".join(parts)

sql = sql1 + sql + endsql

print(sql)

cursor.execute(sql)
```

We use the **cursor.rowcount** function to verify the query or command worked. If it returns a "0", something went wrong. Anything else means success:

```
print(f"Cursor.rowcount: {cursor.rowcount}")

connection.commit()

print(f"Cursor.rowcount: {cursor.rowcount}")

lastrow = cursor.lastrowid

print(f"{lastrow=}")
```

Here at the end, of the function, we refresh the entire recordset, load the form, reset the TEntry widgets to disabled (to keep the user from accidentally changing something), and then disable the save button and re-enable the movement buttons:

```
get_emps()

load_form("id", which)

setup_for_edit(True)

_w1.btnSave.configure(state=DISABLED)
```

```
    enable_record_move()
```

The **write_full_record**We use the same insert SQL statement that we did in the utility program that made the database:

```
def write_full_record():
    fn = _w1.EntryFirstName.get()
    mn = _w1.EntryMiddleName.get()
    ln = _w1.EntryLastName.get()
    add1 = _w1.EntryAddr1.get()
    add2 = _w1.EntryAddr2.get()
    if add2 == "":
        add2 = " "
    city = _w1.EntryCity.get()
    state = _w1.EntryState.get()
    postal = _w1.EntryPostal.get()
    dob = _w1.EntryDOB.get()
    phone = _w1.EntryPhone.get()
    dept = quote(_w1.combobox.get())
    print(fn, mn, ln, add1, add2, city, state, postal, dob, phone, dept)
    # Get department Primary key
    deptID = get_department_id(dept)

    sql2 = """INSERT INTO "Employees" (FirstName,MiddleName,LastName,Ad-
    dress1,Address2,City,State,PostalCode,DateOfBirth,PhoneNumber,De-
    partmentID,active) VALUES (?,?,?,?,?,?,?,?,?,?,?,?);"""
    info = (fn, mn, ln, add1, add2, city, state, postal, dob, phone,
    deptID, 1)
    try:
        cursor.execute(sql2, info)
```

```
            connection.commit()

            success = cursor.rowcount()

            lastrow = cursor.lastrowid

            print(f"{lastrow=}")

            get_emps()

            load_form("actual", shared.actual_records)

            if lastrow == 1:

                titl = "Database Viewer"

                msg = "Write to database successful"

                messagebox.showinfo(titl, msg, parent=_top1,
                icon=messagebox.INFO)

        except:

            titl = "Database Viewer"

            msg = "Failed to write to database"

            messagebox.showerror(titl, msg, parent=_top1, icon=messagebox.
            ERROR)
```

The `on_TCheck` function is the callback whenever the *TCheckbutton* is clicked. We set the shared `showDeleted` flag properly, clear the form, get all the employee records again based on the state of the `showDeleted` flag, load the form, set the state of the Checkbutton into the configuration file, and update it. A whole lot of things for such a simple and short function:

```
def on_TCheck(*args):

    tf = _w1.tch47.get()

    if tf:

        shared.showDeleted = True

    else:

        shared.showDeleted = False

    clear_form()

    get_emps()
```

```
    load_form("id", 1)

    shared.config_data["Startup"]["showdeleted"] = shared.showDeleted

    update_config_data()
```

The `on_btnUndoDelete` function is a callback for the *UndoDelete* button. Basically, it gets the data for the record, verifies that the user wants to "undelete" the record and if so, sets the Active field to 1 and commits the change. If the user decides not to, the undelete function is canceled:

```
def on_btnUndoDelete(*args):

    global emprecs  # currentrec

    trec = list(

        cursor.execute(f"SELECT * FROM Employees WHERE EmpID={shared.
        currentrec}")

    )

    if trec[0][12] == 0:

        titl = "Database Viewer"

        msg = "This will restore this record to Active Status.  Proceed?"

        resp = messagebox.askyesno(titl, msg, parent=_top1,
        icon=messagebox.QUESTION)

        if resp:

            sql = f"UPDATE Employees SET Active=1 WHERE EmpID={shared.
            currentrec}"

            cursor.execute(sql)

            print(f"{cursor.rowcount=}")

            connection.commit()

            print(f"{cursor.rowcount=}")

            if shared.showDeleted:

                sql = """SELECT *, COUNT(*) OVER(ORDER BY EmpID) as
                RecNumber

        FROM Employees"""
```

```
            else:

                sql = """SELECT *, COUNT(*) OVER(ORDER BY EmpID) as
                RecNumber

        FROM Employees where Active = 1"""

            emprecs = list(cursor.execute(sql))

            load_form("id", shared.currentrec)

    else:

        titl = "Database Viewer"

        msg = "Delete operation CANCELED!"

        resp = messagebox.showinfo(titl, msg, parent=_top1,
        icon=messagebox.INFO)
```

The `get_department_id` function is simply a helper function. Since the *TCombobox* holds just the department names, we need to query the Department table with the name to get the record ID in order to save or update the records:

```
def get_department_id(deptname):

    sql1 = f"""SELECT DepartmentID FROM Department WHERE DeptName =
    {deptname}"""

    print(sql1)

    result = list(cursor.execute(sql1))

    print(result[0][0])

    return result[0][0]
```

The `on_btnCancelEdit` function is another simple function that changes the state flags of the `btnEdit` and `btnCancelEdit` buttons and resets the form back to the "viewer" mode:

```
def on_btnCancelEdit(*args):

    _w1.btnUnEdit.config(state=DISABLED)

    _w1.btnEdit.config(state=NORMAL)

    enable_record_move()

    shared.edit mode = ""
```

```
    setup_for_edit(True)
```

Finally, we get to the **set_icon**This is a purely decorative function that sets an icon for the form. In Linux, it only sets an icon on the panel. In Windows, it sets the icon on the window form itself:

```
def set_icon():

   global progImagesPath

    img = os.path.join(location, "graphics", "database.png")

    shared.p1 = tk.PhotoImage(file=img)

    root.tk.call("wm", "iconphoto", root._w, shared.p1)
```

Conclusion

We have flown through creating a front-end form for SQLite databases. You can also use the information presented to create front ends for databases like MySQL, SQL Server, and just about any other database engine. Many of the statements will change, of course, since SQLite uses a subset of the full SQL definition.

I have shown, how to use the SQLite statements to create, fill, update, insert, and generally get information from the SQLite database system. It would be wise if you spent a good amount of time looking at the SQLite home page website at **https://www.sqlite.org/index.html**. There you can get much more in-depth on the SQL statements.

In our next chapter, we will take an in-depth look at the Preferences menu, how to change things to suit your design style, and how to create a custom profile (or multiple profiles) to help you with your GUI design process.

Multiple Choice Questions

1. How do you cause the Toplevel form to use a theme for your project?
   a. style.theme_create({ThemeName})
   b. style.theme_names({ThemeName})
   c. style.theme_use({ThemeName})
   d. style.theme_settings({ThemeName})

2. How do you get a list of the Themes available?
   a. style.theme_create({ThemeName})
   b. style.theme_names()
   c. style.theme_use({ThemeName})
   d. style.theme_settings({ThemeName})
3. What is the SQL statement to create a database?
   a. "CREATE TABLE ..."
   b. "CREATE DATABASE ..."
   c. Nothing. Just set a connection to the filename of the database.

Answers

1. c
2. b
3. c

Questions

1. How do you load a theme into your project that is not installed in Python?
2. What is the SQL statement for updating a record?
3. How do you disable a TEntry widget?

Answers to the questions

1. You should use the Tcl function call:

```
_top1.tk.call("source", os.path.join(location, {Theme_Folder},
{ThemeName}))
```

2. `"UPDATE {Tablename} SET {fieldname} = {value} WHERE {condition}`
3. `"{TentrywidgetName}.config(state="disabled")`

Key terms

- data record
- data field
- primary key

- SQLite

- Connection
- Cursor
- Commit
- SQL statement
- Insert into
- update
- Select
- Database

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10
Creating Custom Profiles

Introduction

Most users never look at the preferences menu, much less make any changes. This is not necessarily a bad thing on the face of it. If the "generic" settings for PAGE work for you, then why would you even consider changing things? As the saying goes, *"If it ain't broke, don't fix it."* From a tech support view, the fact that most users do not even realize there is a way to change things is a good thing.

However, if you want to take the power user point of view, you will want to know about the preference menu options and how to take advantage of them. One important thing to remember about making any changes to any of the preferences there are a few things you need to know BEFORE you start digging into the preferences.

In this chapter, we will examine the options for the Preference menu, why you would want to change it, and how to do it properly.

What you will learn

In this chapter, you will learn the following topics:

- How to access the Preferences Menu
- Available options in the Preferences Window
- How to create Custom Preferences
- How to make a Custom Preference the Default

Getting started

The most important thing about doing anything to the PAGE preferences is that you do not want to make any changes to preferences when you are in an active project. This could cause your project to get messed up.

Many users start PAGE simply by calling PAGE from the terminal (command prompt) without supplying a project name. This is completely fine. However, if you know what your project will be called, you can start PAGE by providing a project name as follows:

```
page MyNewProject.tcl
```

or simply:

```
page MyNewProject
```

This starts PAGE ready to save any information to the .tcl file, which is how PAGE sees the layout of your project. Once you get to a point that you want to save, simply select **File** | **Save** (or *<Ctrl> + S*) to save the project file. Once you get to a place where you want to rename your project to a different name, all you need to do is do a **File** | **SaveAs** (or *<Ctrl> + A*) and enter the new name for your project as shown in *Figure 10.1*:

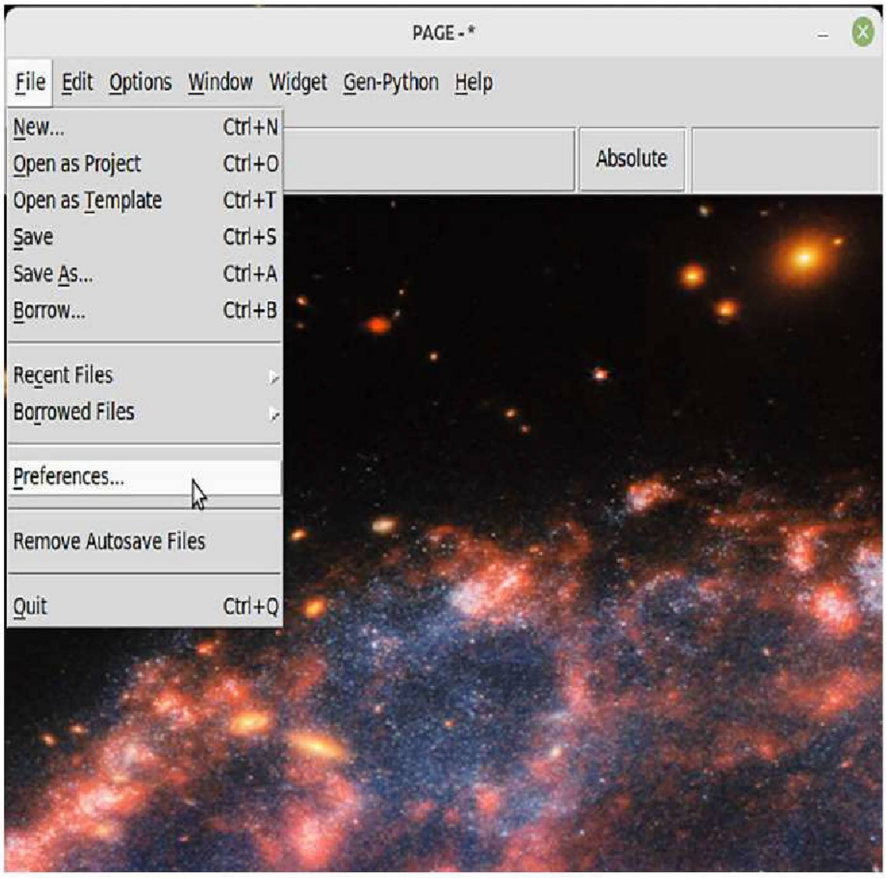


Figure 10.1: *Accessing the Preferences Window*

When you want to look at the Preferences and especially if you want to make any changes, always start PAGE without providing a project name. I cannot stress this point enough. If you do not follow this rule, you run the risk of not only messing up your preferred file but also running the risk of messing up your project file as well. If the project file gets messed up, there is very little that you can do to fix it other than going back to a saved backup for the project and you will potentially lose a lot of the changes you have made.

Assuming that you followed the rule, this is what the PAGE preference window will

look like:

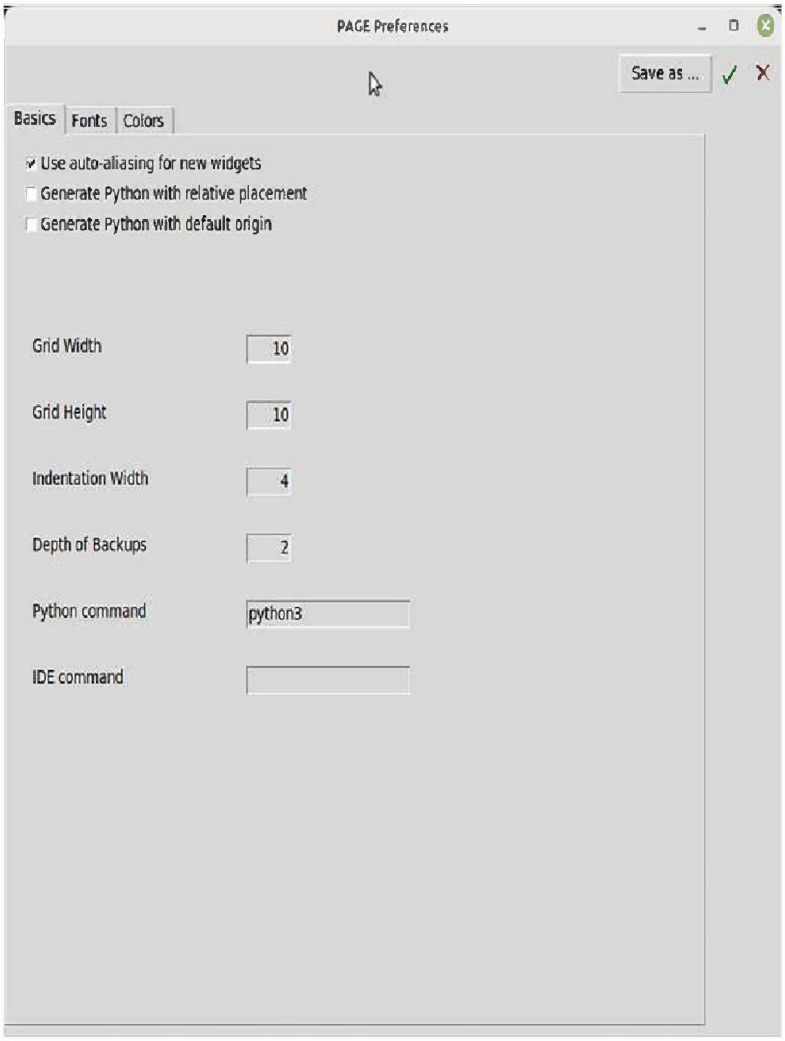


Figure 10.2: Preferences Menu

You will notice that there are three tabs for the Preference window. The first is for basic or general preferences, one for the Fonts used by PAGE, and one for the colors.

We will start with the Basics tab.

The Basics tab

At the top of this tab, there are three checkbuttons, one that specifies whether PAGE will provide an automatic alias when you place a widget on your Toplevel form, which should always be checked, one that controls whether you are going to be in the Relative or Absolute placement mode when PAGE starts a new project, and one that controls if PAGE will write the Python files with a default origin. This should always be unchecked.

Before we go any further, let us take a look at the middle or second checkbutton, the one controlling the Relative or Absolute mode because both modes have pluses and minuses. When you design in the Absolute mode (second checkbutton is unchecked), when you place a widget on your Toplevel form, PAGE uses the exact X/Y position and the exact width and height of the widget when it generates the Python GUI file. If you are very picky about how your form looks, this Absolute mode is what you want. That is the plus. The minus of using the Absolute mode is that the user cannot resize the Toplevel form when it gets run.

For the Relative mode, the placement and size of the widgets are calculated as a percentage of the size of the container. If you are placing a widget directly on the Toplevel form, that is the container of the widget, the size of the form (both at design time and at run time) is considered when placing and sizing the widget itself. On the other hand, if your widget is going into a container like a Frame or Labelframe (or even a canvas widget) when the form is resized so is the container widget. Therefore, the position of the widget within the frame will change. If you have spent hours on your layout, getting things just right and you are in the Relative mode, your careful spacing will change as will the look of your form.

Moving on to the configurable values portion of the form, the first two control the invisible grid that PAGE uses to place your widgets. By default, this is set to a 10x10 grid. I have never had a reason to change this, since I tweak the position of each widget as I place them and other widgets. Be careful if you consider changing these two settings.

Indentation width controls the number of spaces that will be used when creating both the GUI and support Python files. The default of 4 spaces is the "general" suggested indent spacing for Python files. That having been said, it is not required to keep it this way, but I would suggest that you think "thoroughly" about changing this value.

The depth of backups controls the number of backup files that will be created in the

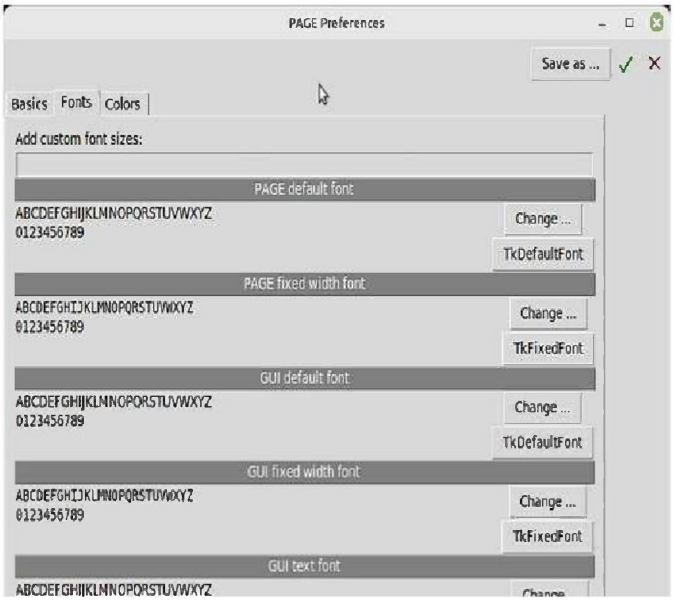
"backups" folder. Usually, the default of two is sufficient; however, you can safely change it to a higher number if you have a good amount of free disk space on your development machine. I would not suggest lowering this value below the default of two.

Python command is the command that will be used to start Python not only when you want to do a preview run by selecting the "run" button in the PAGE GUI and Support modules, but also for the hashbang lines at the beginning of each of the Python files.

The IDE command field allows you to provide the startup file command for your IDE of choice. This can then be used by selecting "Gen-Python| Load project into IDE" from the PAGE Main Window. If, for example, you use IDLE as your IDE and you start it by typing "idle3" in your terminal or command prompt, you can simply type "idle3" in this field. Different IDEs take different command line text, so check the Internet for the command line you need to use for your IDE. Usually, it is simply the name of the program. I use Visual Studio Code as my IDE so I can put "code" in this field.

Font tab

The second tab is for configuring the fonts that PAGE uses for both itself and for your projects as shown in *Figure 10.3*:



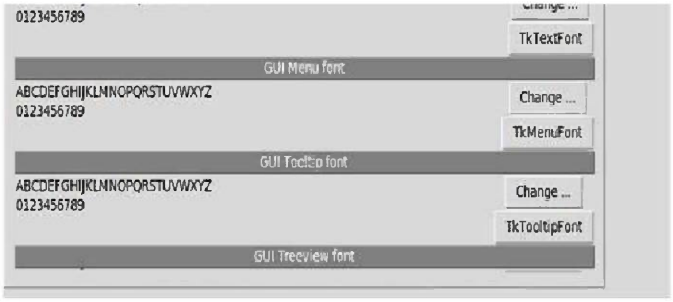


Figure 10.3: Preferences Font Tab

Looking carefully, you will see that while there are many items here, they are (somewhat) grouped into two groups. One for the fonts that "belong" to PAGE and its windows and the other for the generated GUI files.

You absolutely may change the look and feel of the PAGE windows. This includes the Main window, the Toolbox, the Widget Tree, and the Attributes Editor. This tab only controls the fonts, but the Colors tab controls the colors.

With each option, there are two buttons on the right side. One marked "`Change ...`" and the other which starts with "`Tk`" which resets the font for that item back to it's default, just in case you don't like the changes you've made.

At the very top of the tab, just under the tabs themselves is a line that says **`Add custom font sizes:`** and a small gray area which is an Entry widget. It is somewhat hard to see under the standard setup. Here, you can enter extra font sizes that will be included in the Font Selection dialog box as shown in *Figure 10.4*:

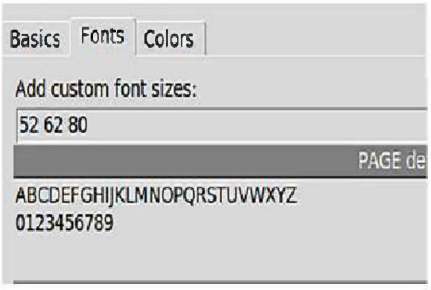
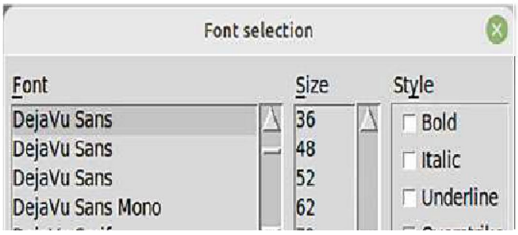


Figure 10.4: Custom Font Sizes

The default `Font Selection` dialog looks like this:



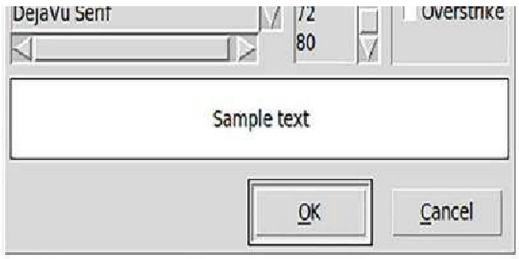


Figure 10.5: Default Font Selection Dialog

You can see that the default font sizes go up to `30`, then jump to `36`, `48`, and `72`. Many users requested that PAGE supports larger sizes than `72` points since they use PAGE to design Kiosk applications. Using the custom font size feature will add the values you want and need, once you saved the preferences and restarted PAGE as shown in *Figure 10.6*:

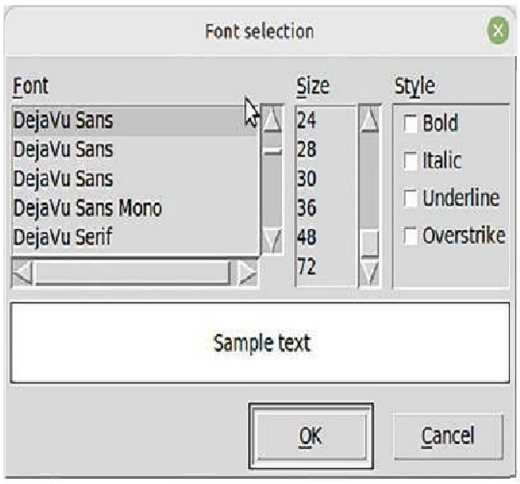


Figure 10.6: Custom Font Sizes Added

We will look at saving the preferences shortly.

Colors tab

The third tab allows you to configure the colors that PAGE uses both for itself and for your project widgets:

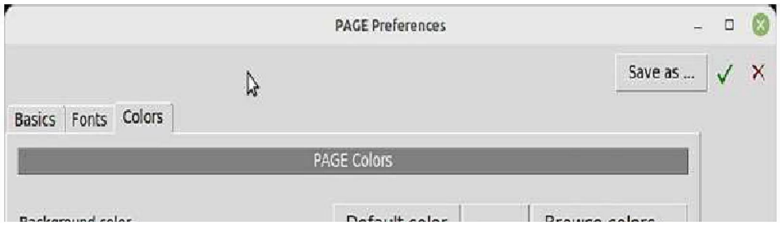


Figure 10.7: Preferences Window Colors Tab

On the right-hand side of the Preferences window, there are two buttons and a gray or black rectangle between them. The button marked *Default color* resets the value for that option. The button marked **`Browse colors`** will bring up the color picker dialog. The colored rectangles will reflect a visual indication of the color that is currently in use for that option. The default color picker dialog looks like this:

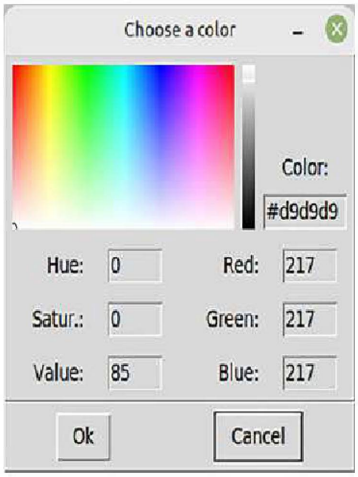


Figure 10.8: Default Color Picker

There are many ways to select a color. You can click inside the color area and on the gradient strip on the right of it to change colors. You can manually change the values in the Red, Green, and Blue Entry widgets, you can manually change the values in the Hue, Saturation, and Value boxes, or you can manually enter the hex value directly into the color Entry box. You can even use valid Named colors like **`wheat`** or **`seagreen3`**. Once you have made your selection, simply click on the **`Ok`** button or

`Cancel` to forget about it:

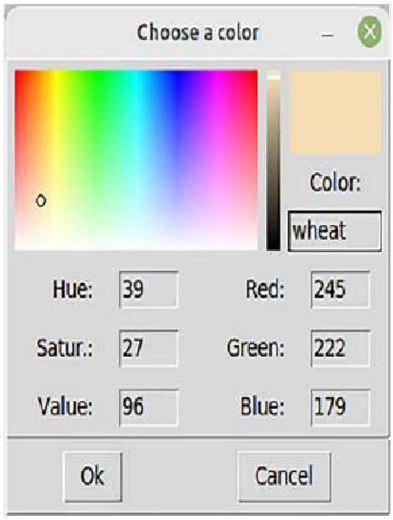


Figure 10.9: Color Picker with text in Color entry

Let us assume you want to have the `wheat` color in all the background options:

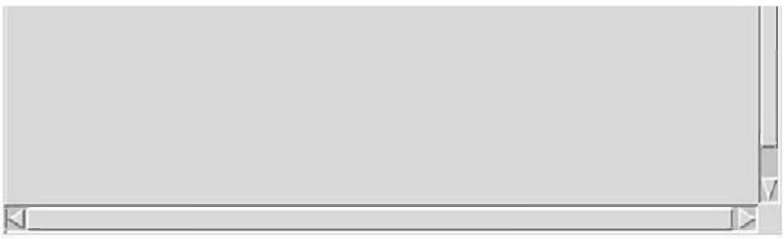


Figure 10.10: Color Tab after changes

This will change all the background colors for PAGE, the GUI that you design, and for the menus. Once you save the settings, when you restart PAGE, the PAGE main menu and all other objects will look something like this:

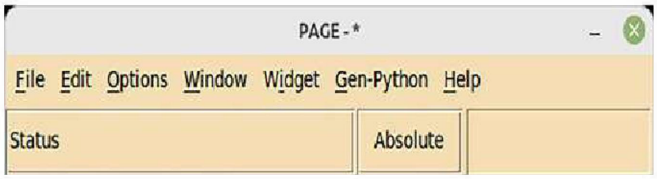


Figure 10.11: PAGE Main menu using Wheat color

Saving the preferences

Now, if you have made any changes, you will probably want to save the changes so that they will be used in PAGE when you start it up next time. There are three options here. The first is to discard all changes and simply close the window. If that is your decision, click on the "X" in the upper right-hand portion of the Preferences window. (NOT the X to actually terminate the window on the title bar.) Your second option is to save the changes and default to PAGE. This overwrites the normal PAGE preferences file (.pagerc). To do this, click on the green check in the upper right-hand corner of the preferences window. The final option is to save this as a custom preference file. This will keep your default preferences file safe and you can start PAGE with the normal look and feel that you are used to, but will apply your customizations by telling PAGE to load the custom preference file when you start PAGE. We will talk more about that in a little bit. Right now, either discard the changes or click on the green checkmark to save the changes. This will save the changes and close the dialog. You should restart PAGE after you change the preferences.

Creating a custom preference profile

Creating a custom preference is almost just as easy as changing your standard profile. Follow the same steps for changing your profile except to save the changes click on

the Save As button instead of the green check mark:

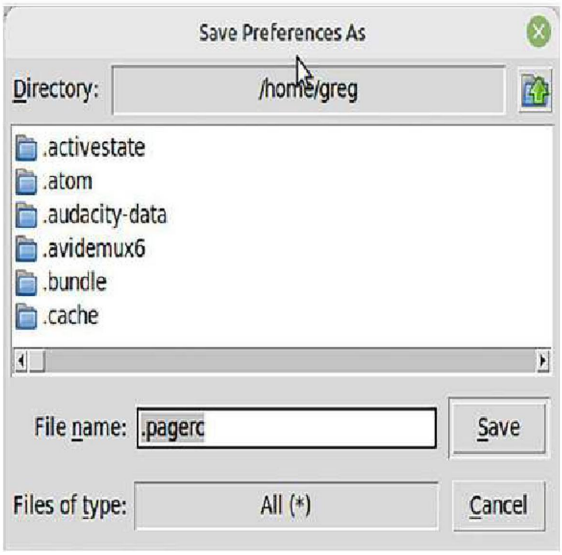


Figure 10.12: Preferences Save As Dialog

You should save your custom preference profile in the same place as the standard PAGE profile, which is the home folder. You should also begin the name with a period "." like the default PAGE profile. So, if you need to call your custom profile mydarkprofile, you should use something like "`.mydarkpagerc`" just to be sure that it will not be confused with any other files on your computer.

In order to tell PAGE to use your special profile, you start the page like this:

```
page -p <rc_file>
```

or to include a project name:

```
page -p <rc_file> {project_filename}
```

You ***MUST*** provide your custom profile name before you provide your project name. You will want to start PAGE this way whenever you are working on this project to make sure your preferences stick when making changes to your forms.

Using the custom resource file also has the added benefit of letting you test your changes without committing to overwriting your "normal" preferences file. Once you have verified you like these changes, then open PAGE without a project using the -p option, open the Preferences window again, select the Save As button, and change the filename from your custom file to "`.pagerc`". This should commit any changes

to the `.pagerc` file. From then on, when you start PAGE, the custom preferences will be the default.

Conclusion

In this chapter, we covered using the Preferences dialog of PAGE.

In the next chapter, we will look at using one of the most complicated but flexible widgets that PAGE and Tkinter offer, the Canvas widget.

Multiple Choice Questions

1. How do you get to the Preferences Window?
   a. From the Edit Menu item.
   b. From the File Menu item.
   c. From the Window Menu item.
2. How many option tabs are there in the Preference Dialog?
   a. 1 b. 2 c. 3 d. 4

3. If you need to save the preference profile for normal PAGE use, you should:
   a. Click on the green check mark
   b. Click on the X.
   c. Click on the Save As button.
   d. Just close the window.

Answers

1. c
2. c
3. a

Questions

1. Are there any Preferences options that should not be changed?
2. Is there a way to change the font size dialog?
3. Can I change preferences in the middle of a design session?
4. What do I do if I mess up my preferences file?

Answers to the questions

1. Yes. While you can change all the options, there are a few you should not change unless there is a very good reason to. These options are "Use auto-aliasing for new widgets", "Generate Python with default origin", "Grid Width", and "Grid Height".
2. Yes, at the top of the Fonts tab, there is an Entry box where you can specify additional font sizes. Font sizes should be entered without commas between the entries. For example, if you want to add font sizes 52, 62, and 80, enter them as 52 62 80. Be sure that you save the preferences by clicking on the green check mark for normal preferences or the Save As button for custom preferences.
3. While the answer is Yes, you risk corrupting both the preferences file as well as your project. Please do not.

   Start PAGE without a project name, immediately go to Preferences, make and save your changes, and then exit PAGE.
4. Simply delete the profile from your computer then restart PAGE. The page will regenerate a default preferences file on startup.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 11

Using the Canvas Widget

Introduction

In this chapter, we will explore many of the ways that you can use the Canvas Widget, one of the most powerful widgets in the Tk/Tkinter toolkit.

Many users, especially new users do not know how to use Canvas, other than a simple container. However, this is just the tip of the iceberg.

We will be working on two projects in this chapter. First, we will create an application that shows some of the various methods that the Canvas widget has as shown in *Figure 11.1*:

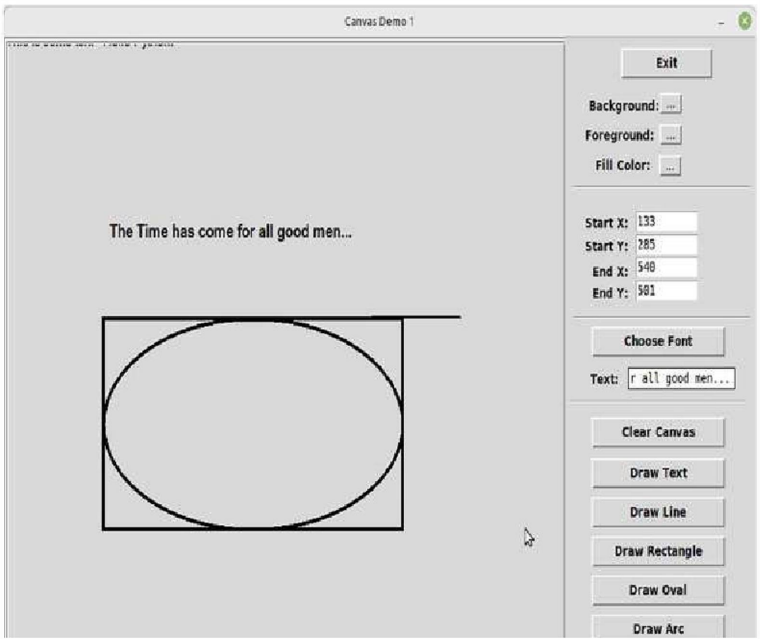


Figure 11.1: Canvas Demo

In the second project, we will create a simple animated sprite program showing a walking dinosaur and moving clouds:

Figure 11.2: Animated Sprite

What you will learn

In this chapter, you will learn the following topics:

- Using the `create_text`, `create_line`, `create_oval` functions and more
- Creating animated screens for your application

Project #1 – Canvas functions

As we just saw, the Canvas widget is more than just a container; many functions allow us to generate graphics dynamically. These methods include `create_arc`, `create_image`, `create_line`, `create_oval`, `create_polygon`, `create_rectangle`, and `create_text`. There are two other methods that we will not explore here which are `create_bitmap` and `create_window`. There is also a `create_image` method that we will look at in the second project.

Here is what the project looks when it first starts up:

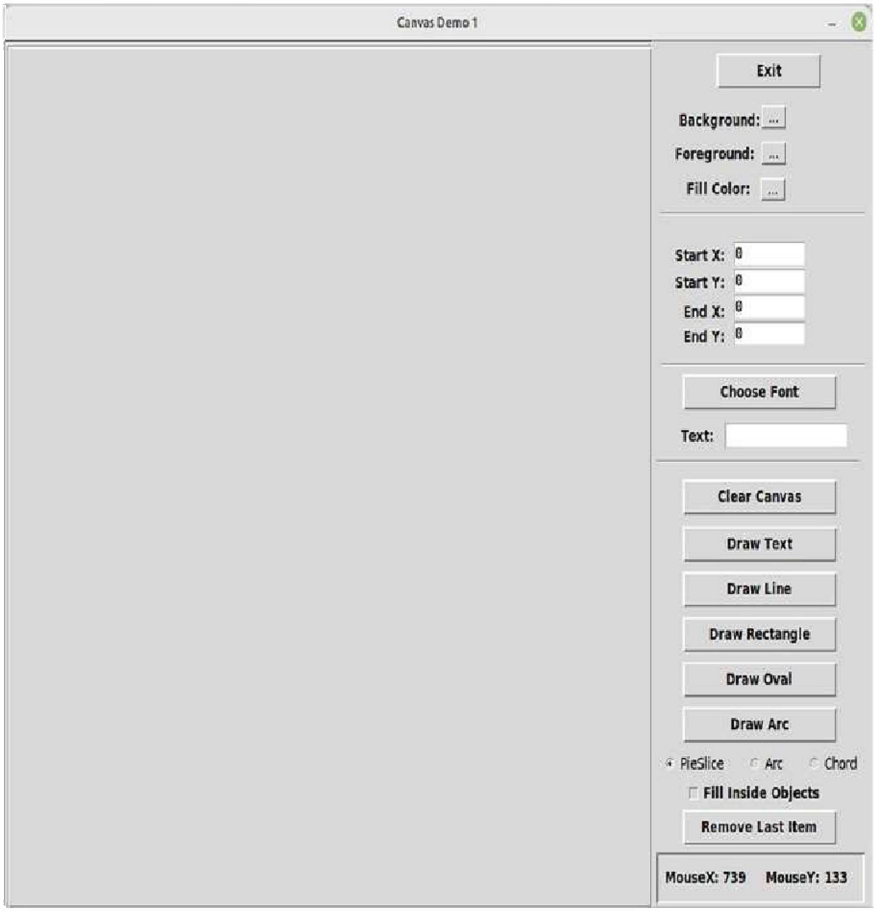


Figure 11.3: Canvas Demo at Startup

You can see that the form itself is fairly straightforward but contains many widgets. On the left is the canvas widget and on the right are several controls that will allow us to demonstrate the various methods and support options. For example, we have three buttons at the top right of the form marked with "...". These, when clicked will provide a color picker dialog for the user to choose a background color (for the Canvas itself), a foreground color, and a fill color. Just below that are four entry widgets that show the values for start x and y and end x and y for drawing (or creating) the various graphic objects.

Below, there is a button to display a font picker and an entry widget for the user to define text for the **create_text** method.

Further down are 6 buttons that will clear the canvas and call the various methods that the Canvas widget supports. Below the Draw Arc button are three Radiobuttons that provide the various options for the **create_arc** method. Below that is a Checkbutton that, when checked, will cause the objects (when supported) to be shown with the fill color. One last button provides a way for the last object drawn to be removed. Finally, there is a "status bar" that shows a real-time value for the x and y position of the mouse within the Canvas widget.

Using the program is straightforward. The Mouse-1 button (your index finger button) sets the start X and Y positions. The Mouse-3 button (your ring finger button) sets the end X and Y positions for lines, rectangles, ovals, and arcs. The Mouse-2 button (middle mouse button) can be used to add extra points to draw polygons (objects with more than a start and end position).

Since there is a large amount of code for this and project #2, I will not go through the design of the project. You can find it in the provided examples under the folder for *Chapter 11 Using the Canvas Widget* under the *Demo1* folder. Also, I will not provide the entire code for this project here in the chapter. Instead, I will concentrate on various functions to show how to use the Canvas Methods.

Code for Project #1

As always, we need to start with the import section. Any code that needs to be added will be in bold. For this project, we need to include the Python time and math modules as well as the **colorchooser**, font, and **messagebox** modules from Tkinter:

```
import sys

import time

import math
```

```
import tkinter as tk

import tkinter.ttk as ttk

from tkinter.constants import *

from tkinter import colorchooser

from tkinter import font

import tkinter.messagebox as messagebox
```

As we have done with just about every project, we need to create a startup function and call it from the main function that PAGE provides for 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 = CanvasDemo.Main(_top1)

    startup()

    root.mainloop()
```

In this project, the startup function deals with many things, not the least of which is initializing various global variables. I will only present some of these functions to save space:

```
def startup():

...

# Bindings

    _w1.Canvas1.bind("<Motion>", motion)

    _w1.Canvas1.bind("<Button-1>", set_start)

    _w1.Canvas1.bind("<Button-2>", set_extra)
```

```
    _w1.Canvas1.bind("<Button-3>", set_end)

    _top1.bind_all("<KeyRelease>", on_keypress)

...
```

We need to set up bindings for the Canvas widget since it does not have a command method and because we need different mouse events. These are for motion, Mousebutton-1, Mousebutton-2, and Mousebutton-3. We also need to provide a key event to support the "hidden" options. We have seen the Tk bindings before.

We also keep a list of objects that have been created so they can be deleted later on. This list is called a **stack**. Each method that we use from the Canvas widget provides a return value, which is the object.

The program

I decided to provide a red line that runs between the start X and Y position which is set by Mousebutton-1 and the current mouse position. This is called a `rubberband` since it stretches and moves to the current mouse position:

```
    global rubberband, rb1

    rb1 = None

    rubberband = False
```

The `rubberband` is controlled by the motion event since we need to know the current X and Y positions. Here is the motion Callback function:

```
def motion(event):

    global mousex, mousey

    x, y = event.x, event.y

    mousex = x

    mousey = y

    show_xy()

    global rubberband, rb1

    if rubberband:

        if rb1 != None:

            _w1.Canvas1.delete(rb1)
```

```
        x0 = _w1.StartX.get()

        y0 = _w1.StartY.get()

        rb1 = _w1.Canvas1.create_line(x0, y0, mousex, mousey, fill="red")
```

The callback is straightforward. We get the current X and Y positions from the event data sent in as a parameter to our callback. Once we have them, we call the **show_xy** function to display the mouse position in the statusbar. We then check to see whether we have the StartX and StartY positions set and if so, we remove the current red line from the Canvas widget, then redraw a new red line from the StartX and StartY positions to the new mouse position. We use the **create_line** method to do this.

```
rb1 = _w1.Canvas1.create_line(x0, y0, mousex, mousey, fill="red")
```

To draw text, there is a command callback linked to the Draw Text button. We have to do some tasks like remove the start and end marks if they exist then check to see whether the user has entered a string of text in the Entry widget to be displayed. If not, use the default text string. The actual call to display text on the Canvas widget is straightforward. Finally, we add the object tag to our object stack so it can be removed later:

```
def on_DrawText(*args):

    global backgroundColor, foregroundColor, fillcolor, displayfont

    global startmark, endmark, objects, defaultTextString

    ...

    if (_w1.StartX.get() != "") and (_w1.StartY.get() != ""):

        if startmark:

            _w1.Canvas1.delete(startmark)

        if endmark:

            _w1.Canvas1.delete(endmark)

        if _w1.TextData.get() != "":

            text = _w1.TextData.get()

        else:

            text = defaultTextString
```

Now that the housekeeping is done, we can make a call to the `create_text` method. The syntax for `create_text` is:

```
objname = create_text(startX, startY, fill=fillcolor,
font=font,text=text,anchor)
```

So, we place the on the canvas and then we add it to the object stack with the append call:

```
        textobj = _w1.Canvas1.create_text(
            _w1.StartX.get(),
            _w1.StartY.get(),
            fill=foregroundColor,
            font=displayfont,
            text=text,
            anchor="w",
        )
        objects.append(textobj)
```

Tkinter exposes a number of utility functions like message boxes, file dialogs, and color pickers but it does not give any access to the Tk font chooser. However, we can make a direct call to it. We can use the `tk.call` method like this:

```
def fontChooser():
    root.tk.call(
        "tk",
        "fontchooser",
        "configure",
        "-font",
        "arial 10",
        "-command",
        root.register(fontChanged),
    )
```

```
    root.tk.call("tk", "fontchooser", "show")
```

Then, the callback function, **fontChanged**, will then take the information and use the newly selected font:

```
def fontChanged(newfont):

    global displayfont

    displayfont = newfont

    print(newfont)
```

We can clear the entire canvas by using the delete method. The standard delete method takes either the object name or **all**:

```
def on_btnClearCanvas(*args):

    if _debug:

        print("CanvasDemo_support.on_btnClearCanvas")

        for arg in args:

            print("    another arg:", arg)

        sys.stdout.flush()

    _w1.Canvas1.delete("all")
```

The create_line method syntax is very similar to the **create_text** method. However, there is not only a **startX** and **startY** position but an **endX** and **endY** position as well:

```
def on_drawLine(*args):

    global backgroundColor, foregroundColor, fillcolor, displayfont

    global startmark, endmark, objects

    if _debug:

        print("CanvasDemo_support.on_drawLine")

        for arg in args:

            print("    another arg:", arg)

        sys.stdout.flush()

    if startmark:
```

```
_w1.Canvas1.delete(startmark)
```

```
if endmark:
    _w1.Canvas1.delete(endmark)
doFill = _w1.che47.get()
x0 = _w1.StartX.get()
y0 = _w1.StartY.get()
x1 = _w1.EndX.get()
y1 = _w1.EndY.get()
lineobj = _w1.Canvas1.create_line(
    x0, y0, x1, y1, fill=foregroundColor, smooth=True, width=3
)
objects.append(lineobj)
```

In our project code, we cover most of the other methods that the Canvas widget has. But now, you can begin to see the potential of what the Canvas widget can provide. In our next project, we will take a look at the `create_image` method of the Canvas.

Project #2 – animated canvas

In this project, we will look at using the `create_image` method of the Canvas widget to do graphic animation as shown in *Figure 11.5*:

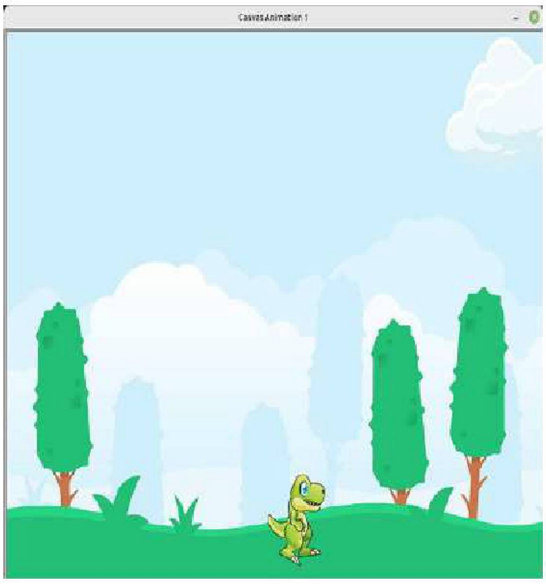


Figure 11.4: Sprite Animation

Many games use methods similar to the ones I will show you, to make games. Most of the games that are written in Python use a library called PyGame, which is a fantastic add-on to Python to do many things. However, I wanted to go through the basics with no gaming add-ons, just using the Canvas widget. The code for the project is located in the sprite folder for *Chapter 11*.

The clouds and background are from **https://kenney.nl** and the character images are from **https://www.gameart2d.com/free-dino-sprites.html**. Both sets are free for anyone to use.

To use the character images, I needed to resize them from the original 680 pixels by 472 pixels down to 192 pixels by 133 pixels. In addition, the original images are all in one folder and for my design to work, I needed to them be in separate folders, one for walk and one for jump. Finally, I needed to flip the images since all of the original images face right and I need left-facing images as well. I used the Gimp image program to handle this. I also created a folder for the Clouds and copied them from the distribution folder to my folder.

How it works

After setting various variables in the startup function, we call a function called `load_images()`. I create several dictionaries to hold, each of the images, one for the sprite `walking right`, one for walking left, one for jumping right, one for jumping left, and one for the clouds. Each dictionary is then loaded by using the glob method of Python which will recursively find all the filenames in a folder that matches a pattern. In this case, the pattern is `*.png`. Since all the images for the character walking right are in the folder `DinoR`, glob returns the filenames for all the filenames of the dinosaur walking right and each of those will be used to create an image object which is then loaded into the various dictionaries. There is a block of code like the following one for each image set:

```
# Load the character walking Right

imgdir = os.path.join(_location, "graphics", "DinoR")

for f in glob.glob("%s/*.png" % imgdir):

    img = os.path.split(f)[1]

    name = img[:-4]

    imgsR[name] = tk.PhotoImage(name, file=f)
```

Once we have the dictionaries, we go through and create a list of the keys for the dictionaries. Again, there is a block of code similar to the following one for each image set:

```
keysR = list(imgsR.keys())

keysR.sort()
```

We also need to bind all the keypress animations to the entire Toplevel form:

```
_top1.bind_all("<KeyRelease>", lambda e: on_keypress(e))
```

Once all the image sets are loaded, we start two animation loops using `root.after`:

```
global cloudsId, animateId, ignoreKey

ignoreKey = False

animateId = root.after(0, on_animatesprite)

cloudsId = root.after(0, on_animateClouds)
```

If you remember, `root.after` requires an id (the return value) the number of milliseconds to wait until making the callback to the function provided. Since we are setting up the functions for the first time, we call it with a 0 in the wait time parameter, which calls the callback immediately. At the end of callback(s), we will set the delay time. I will only show the animation loop for the character sprite. The animation loop for the clouds is very similar, but in that function, we handle each of the three cloud sets one after the other and the delay time will be set to 200 milliseconds. Since the right facing sprite and left-facing sprite code is very similar, we will only look at the right-facing image code:

```
def on_animatesprite():

    global imgsL, imgsR, cntr, spriteid, xpos, animateId, dir

    global keysR, keysL, action, currentsub, currentxpos

    currentsub = "on_animatesprite"

    if spriteid != None:

        _w1.Canvas1.delete(spriteid)

        time.sleep(0.01)
```

First, we verify that there is already a `spriteid` (the current sprite image). This way, we will not get an error when we try to delete the image from the canvas. Then, depending on the direction (right or left), we create the next image and place it onto

the canvas at the correct X and Y positions. The Y position for the normal animation is always set to `580`. The X position will change:

```
if dir == "R":

    spriteid = _w1.Canvas1.create_image(xpos, 580,
    image=keysR[cntr], anchor="nw")

elif dir == "L":

    spriteid = _w1.Canvas1.create_image(xpos, 580,
    image=keysL[cntr], anchor="nw")

curretxpos = xpos
```

Next, we check the value of the variable `cntr`, which holds which of the 10 images for the sprite walk should be next. So, if the `cntr` variable is equal to `9`, we reset it to `0` otherwise we increment it by 1:

```
if cntr == 9:

    cntr = 0

else:

    cntr += 1
```

Now, depending on which direction, the sprite is supposed to move in this loop, we will either increment or decrement the xpos variable. In our case, the best increment/decrement value is `7`:

```
if dir == "L":

    inc = -7

    xpos -= 7

else:

    inc = 7

    xpos += 7
```

We then will use the move method to move the sprite image either the positive 7 or negative 7, once again dependent on the direction:

```
_w1.Canvas1.move(spriteid, inc, 0)

currentxpos = xpos
```

Finally, we check to see whether the sprite location is off the screen, either right or left and if so, we then change the direction for the sprite. After that, we call the `update_idletaskes`, which gives Tkinter a chance to handle anything that might need to be done, before we take back control. Last but not least, we `root.after` again with a delay of 100 milliseconds:

```
    if (xpos >= 980) and (dir == "R"):
        dir = "L"
        cntr = 0
    elif (xpos <= -80) and (dir == "L"):
        dir = "R"
        cntr = 0
    _top1.update_idletasks()
    animateId = root.after(100, on_animatesprite)
```

Now, we will look at the callback for the keypress binding. When a key is pressed on the keyboard, it gets passed into the callback in the variable e (for the event and can be checked by using `e.keysym`. So, the first thing we need to do is check to see whether the program has already been paused through the global variable `isPaused`. If so, we check to see whether the character is either p or P. If so, we set `isPaused` to False, delete the paused text on the canvas, and restart the animation loop. Otherwise, we need to pause the animation, so we call the `root.after_cancel` function and print `PAUSED!` on the canvas:

```
def on_keypress(e):
    global animateId, dir, isPaused, textid, ignoreKey
    if isPaused:
        if (e.keysym == "p") or (e.keysym == "P"):
            print("Pause")
            font = "{Comic Sans MS} 40 bold"
            if isPaused:
                isPaused = False
                _w1.Canvas1.delete(textid)
```

```
            animateId = root.after(100, on_animatesprite)
        else:
            root.after_cancel(animateId)
            isPaused = True
            wid = _w1.Canvas1.winfo_width() / 2
            textid = _w1.Canvas1.create_text(
                wid, 380, text="PAUSED!", font=font, fill="red",
                anchor="center"
            )
```

Otherwise, we check to see whether the character is in the middle of a jump. If so, we ignore the key:

```
    elif ignoreKey:
        e.keysym = None
    else:
```

Otherwise, we check to see whether the key pressed is something we want to look for. Those keys would be the Left Arrow key, Right Arrow key, space or Up arrow(for jump), I for information, q for quit, h for help or p for pause. We check here for pause as well since it might have been missed the first time. If the key is either a left arrow or a right arrow, we simply set the direction to L or R depending on which it was:

```
        if e.keysym == "Left":
            print("Left Key")
            dir = "L"
        elif e.keysym == "Right":
            print("Right Key")
            dir = "R"
```

If the key was space or the up arrow key (for jump), we set to ignore the key to ignore any further keypresses (while we are in the jump animation routine) to reset it:

```
        elif (e.keysym == "space") or (e.keysym == "Up"):
            # jump
```

```
            print("jump")
            ignoreKey = True
            dino_jump()
            ignoreKey = False
```

If the key is I, then we either clear or show some program information:

```
        elif e.keysym == "i":
            global showinfo, infoid, infosubid
            print("Info")
            font = "Arial 14 bold"
            if showinfo:
                showinfo = False
                _w1.Canvas1.delete(infoid)
                root.after_cancel(infosubid)
             else:
                showinfo = True
                infoid = _w1.Canvas1.create_text(
                    10,
                    340,
                    text="Display Info Started",
                    font=font,
                    fill="red",
                    anchor="w",
                )
                infosubid = root.after(0, on_showInfo)
```

If the key is q or Q for quit, we show a message:

```
        elif (e.keysym == "q") or (e.keysym == "Q"):
            print("quit")
```

```
            resp = showMessage(2)
            if resp:
                root.after_cancel(animateId)
                sys.exit()
```

If the key is h or H, show the help message:

```
        elif (e.keysym == "h") or (e.keysym == "H"):
            print("help")
            showMessage(1)
```

Finally, if the key is p or P, do the pause routines:

```
        elif (e.keysym == "p") or (e.keysym == "P"):
            print("Pause")
            font = "{Comic Sans MS} 40 bold"
            if isPaused:
                isPaused = False
                _w1.Canvas1.delete(textid)
                animateId = root.after(100, on_animatesprite)
            else:
                root.after_cancel(animateId)
                isPaused = True
                wid = _w1.Canvas1.winfo_width() / 2
                textid = _w1.Canvas1.create_text(
                    wid, 380, text="PAUSED!", font=font, fill="red",
                    anchor="center"
                )
```

The jump routine is probably the most difficult to do. The first thing to be done is to pause the animation loop, then save the `xpos`, the direction, and the `cntr` counter variables into temporary variables for use after the jump and set a couple of other variables:

```
def dino_jump():

    print("into Jump")

    global imgsJL, imgsJR, keysJL, keysJR, cntr, cntr1, djspriteid,
    animateId

    global dir, lastxpos, currentxpos, xpos

    # Pause the Dino walk animation

    root.after_cancel(animateId)

    # Save the position and direction

    lastxpos = xpos

    lastdir = dir

    lastimg = cntr
```

Next, we need to start the animation. There are 12 different images for the jump sequence. So, we need to deal with each one:

```
    # Now do our jump animation

    if spriteid != None:

        _w1.Canvas1.delete(spriteid)

        time.sleep(0.01)

    lowpos = 580

    direction = "up"

    for cntr1 in range(12):
```

We check to see whether the new sprite ID is not equal to None. If it is, then delete the current image. Then, check to see whether the direction is towards the right or to the left. Then, we get the proper image and put it onto the canvas:

```
        if djspriteid != None:

            _w1.Canvas1.delete(djspriteid)

            time.sleep(0.01)

        if lastdir == "R":

            djspriteid = _w1.Canvas1.create_image(
```

```
            lastxpos, lowpos, image=keysJR[cntr1], anchor="nw"
        )
    elif dir == "L":
        djspriteid = _w1.Canvas1.create_image(
            lastxpos, lowpos, image=keysJL[cntr1], anchor="nw"
        )
```

Now, we do a small sleep and update the Toplevel. Then, depending on the direction that the sprite is facing, set a new position and if the cntr1 < 5 (remember Python is zero-based) the vertical direction should be up otherwise the vertical direction would be down since there are 12 images that we need to step through:

```
    time.sleep(0.01)
    _top1.update()
    if dir == "R":
        if (cntr1 <= 5) and (direction == "up"):
            lastxpos += 10
            lowpos -= 10
            if cntr1 == 5:
                direction = "down"
        elif direction == "down":
            lastxpos += 10
            lowpos += 10
    else:
        if (cntr1 <= 5) and (direction == "up"):
            lastxpos -= 10
            lowpos -= 10
            if cntr1 == 5:
                direction = "down"
```

```
        elif direction == "down":
```



```
            lastxpos -= 10
            lowpos += 10
```

Almost through, we use the move method to move the sprite to the new location:

```
    if dir == "R":
        if cntr1 <= 5:
            _w1.Canvas1.move(djspriteid, 10, -10)
        else:
            _w1.Canvas1.move(djspriteid, 10, 10)
    else:
        if cntr1 <= 5:
            _w1.Canvas1.move(djspriteid, -10, -10)
        else:
            _w1.Canvas1.move(djspriteid, -10, 10)
    time.sleep(0.1)
    _top1.update()
```

Finally, erase the last image, update the Tkinter waiting tasks, set the proper location variables, and restart the animation loop:

```
# Erase the last jump image
time.sleep(0.1)
_w1.Canvas1.delete(djspriteid)
time.sleep(0.05)
_top1.update_idletasks()
# Restart the dino walk animation
xpos = lastxpos
dir = lastdir
cntr = lastimg
animateId = root.after(100, on_animatesprite)
```

```
animateid = root.after(100, on_animatesprite)
```

There are a few other things that are done in the code, but these are the most important functions.

Conclusion

You can see that the Canvas widget is one of the most versatile widgets in the entire Tk/ttk toolkit. Not only can you create games with it, and use it as a container for other widgets, but also create your own graphing display functions as well. The limits are pretty much just your imagination and desire.

Our next chapter is the conclusion.

Multiple Choice Questions

1. If you need to capture keystrokes in your program, what command would you use?
   a. command=getch(e)
   b. bind('<Mouse-1>', on_keystroke)
   c. bind_all('<KeyRelease>,on_keystroke)
2. Which library and function would you use to properly define a graphics folder?
   a. platform.path="mydir/graphics"
   b. os.path.join(location,"graphics","mypicture.png"
   c. "/home/user/desktop/graphics/mypicture.png"
3. Which Canvas method would you use to draw some text?
   a. canvas.draw_text()
   b. canvas.create_text()
   c. canvas.place_text()

Answers

1. c
2. b
3. b

Questions

1. What is the anchor option for the various Canvas methods, and do I need it?

2. Do I need to provide a variable for the id returned by the Canvas methods?
3. How does the create_polygon method work?

Answers to the questions

1. Just like the normal Tk and ttk widgets, the anchor option uses one of the following constants.

   nw, n, ne, e, se, s, sw, w, center

   Using the anchor option for normal Tk and ttk objects, the anchor specifies where the text gets placed within the widget display area. For the Canvas objects, the anchor uses the X and Y location as the position and the anchor is which edge (or center) is located. So, if you have an anchor of nw, then the location X and Y would be the upper left-hand corner of the object.

2. No, the ID returned by the Canvas method does not need to be tracked. However, if you want to move or delete or otherwise manipulate that object later, then you must have it. The bottom line is it all depends on what the ultimate purpose of the Canvas object is.

3. The create_polygon method of the Canvas widget is very similar to the create_line or create_rectangle. Here is the syntax for the method:

   objectID = Canvas.create_polygon(x0,y0,x1,y1,x2,y2...option...)

   where xN and yN are the x and y positions of each point in the polygon. You don't need to specify each directly in the call, you can use a Python list to hold the values and then pass that list as the positional values.

   One thing that must be remembered is that the last location in the positions will be generated to bring the end of the line of the polygon back to the startX and startY positions, creating a closed figure.

   So, if a polygon has the x y positions of 5, 10, 5, 40, 20, 40, 5, 40 when drawn on the canvas, will have a final position of 5, 10 automatically added by tkinter which makes a closed rectangle.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

CHAPTER 12
Conclusion

We have had an incredible journey and I sincerely hope that you feel that you have a better grasp on using PAGE.

Some of the things that you have learned include simply starting PAGE, creating projects with multiple Toplevel forms, using Tk and ttk widgets, and some ways to use standard widgets in different ways. You learned how to create sprite animation in the Canvas widget, how to create Menubars, popup menus, and button bar type menus. Your journey is not over yet, however. The more you use PAGE to create your GUI forms, the more you will learn.

Going further

You should be sure to follow the main PAGE website to be sure to be notified when new versions of PAGE are released. The official webpage is **https://sourceforge.net/projects/page/**.

Check out the PAGE Discussion forum at **https://sourceforge.net/p/page/discussion/**. Some issue reports might be helpful to you.

Resources

Probably the best reference is the New Mexico Tech Tkinter 8.5 reference (NMT). It can be found at **https://anzeljg.github.io/rin2/book2/2405/docs/tkinter/index.html**.

If you are having problems understanding the options available for any Tk or ttk widgets, you can refer to a very helpful website **https://www.tcl.tk/man/tcl/TkCmd/contents.html.**

Every widget (even the ones that PAGE does not support yet) is listed here and while the target audience for this website, you can look and get a description about any of the widgets.

Many times, you might have a problem that the NMT Tkinter reference cannot answer. In this case, the Internet can be your best friend. Simply, do a web search using "Tkinter" and the issue you are trying to find out about.

PAGE has some great resources that come with every distribution file. In the PAGE installation folder, look for the docs folder. There you will see folders for the user manual both in EPUB format and HTML. (The HTML file is also available from the Main PAGE menu under `Help | Open Help`. There is also a *tutorials folder* that has a PDF (over 100 pages), as well as all the code that was used in the tutorial.

Also, in the PAGE distribution is the examples folder. These examples provide the PAGE GUI files, the Python GUI file, and the support file. These examples include things like binding, tooltips, and much more.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

accelerators 154, 155
Adobe Photoshop 134
Attribute Editor 13

B

Basics tab 264-266
Bindings 154
btnCopy function 176
btnCut function 174
btnFileNew function 174
btnHelp function 176
btnInfo function 176
btnPaste function 176
Button widget 66
  attributes 66-68

C

Canvas widget 68, 275, 276
  animated canvas 284-294
  code, writing 278-284
  functions 277, 278
Carriage Return Line Feed (CRLF) character 103
cback method 205
Checkbutton menu items 152-154
Checkbutton widget 68
  attributes 69, 70
check_for_changes function 251
ChildForm 40
clear_form function 246
clear_label function 207
clipboard_get() function 173
CloseMenu callback 171
Colors tab 268-270

Context menus 78
Copy function 171
cursor.rowcount function 253
custom control widget 196
  Plotext 209
  ScrolledCheckedListbox 202
  tkDatePicker 196
custom preference profile
  creating 271, 272
Cut callback function 172

D

DB Browser for SQLite 227
dbutils.py 226
designer form 16
do_languages function 120
do_ping() function 111
DualForms 37

E

embedded function 201
Entry widget 70, 71
Error messagebox 168

F

FileOpen function 175
FileQuit callback 168
FileSaveAs function 175
FileSave function 175
find_by_name function 245
fix_labels function 244
Frame widget 71
Front tab 266-268

G

get_configuration_information
    function 239
get_department_id function 257
getLocalInfo function 104
getPublicIp function 103
Graphical User Interface (GUI)
    forms 133
graphics
  code, writing 142-145
  GUI, creating 136-141

I

installation
  Linux 6
  OSX 7
  Raspberry Pi (Linux) 6, 7
  Windows 6

L

Labelframe widget 73
Label widget 72, 73
lambda function 170
limitsize method 219
Listbox widget 74
load_Data_strings function 124
load_departments function 236, 242
load_form function 239

M

MainForm 37
Menu
  code, writing 166-177
  GUI, creating 155-165
  overview 150

menu items 151
  accelerators 154, 155
  Bindings 154
  Checkbuttons 152-154
  Radiobuttons 151, 152
messagebox 168
Message widget 74, 75
multiple_bar plot function 218
multiple forms 36
  code 44-50
  GUI, designing 36-44
my_styles() function 187

N

notsodark theme 233

O

on_btnAdd function 247
on_btnCancelEdit function 257
on_btnChangeTheme function 244
on_btnDelete function 248
on_btnGetChecks callback
    function 207
on_btnGoFirst 243
on_btnGoLast 243
on_btnGoNext 243
on_btnGoPrevious 243
on_btnSearch function 246
on_btnSwitchMode callback 208
on_btnUndoDelete function 256
on_Marker_Select() callback 217
on_PlotType_select callback 217
on_TCheck function 255
on_time_update function 249
open_database function 242

P

PAGE 1, 11
  support 9
  working with 7, 8, 12
PAGE designer forms 13
  Attribute Editor 13, 14
  code 27-31
  Designer form 16
  GUI design 17-27
  main window 13
  Widget toolbar 15, 16
  Widget tree 15
  window layout 17
PAGE preferences 262-264
  Basics tab 264-266
  Colors tab 268-270
  Font tab 266-268
  saving 271
p.communicate() method 112
Pinger
  GUI, designing 85
  history 84, 85
Pinger GUI
  About form, adding 92, 93
  code 94-130
  designing 89-92
  main menu 85-89
plain global variable 189
Plotext 209-212
  code, writing 213-221
  program, designing 212, 213
Popupmenu widget 78
p.wait() method 112

Q

quality graphics 135
  image formats, converting 135
Quit function 173

R

Radiobutton menu items 151, 152
Radiobutton widget 75
Rapid Application Development (RAD) 1
read_file function 103
RecNumber field 241
requirements
  Linux 5
  OSX 5
  Raspberry Pi (Linux) 5
  Windows 5
root.after() method 119

S

Scale widget 75, 76
ScrolledCheckedListbox 202, 203
  code, writing 204-209
  program, designing 203, 204
ScrolledText box 128
separator 151
set_labels() function 206
set_UI_strings() function 123
setup_for_edit function 237, 238
shared.py 226
showDeleted flag 236, 255
showinfo messagebox 168
Spinbox widget 76, 77
Splash form 50, 51
  code 54-58
  GUI, designing 51-54
SQLeo 227
  URL 227
SQLite databases 227, 228
  code, writing 234-258
  GUI form, creating 232-234
  project database, creating 228-232
start_log() function 119
start_up function 99, 102
Structured Query Language (SQL) 225

T

TButton widget 183
TCombobox widget 190
text conventions
  code blocks 3, 4
  inline references 4
  terminal commands 4
  tips and warnings 4
  widget attributes 4
textvar attribute 22
Text widget 77, 78
The Gimp 134
tkDatePicker widget 196, 197
  code, writing 197-202
  program, designing 197
Tk widgets 62, 63
  Button widget 66-68
  Canvas widget 68
  Checkbutton widget 68-70
  common attributes 63, 64
  Entry widget 70, 71
  Frame widget 71
  geometry 65
  Labelframe widget 73, 74

Label widget 72, 73
Listbox widget 74
Message widget 74, 75
Popupmenu widget 78
Radiobutton widget 75
Scale widget 75, 76
Spinbox widget 76, 77
Text widget 77, 78
Toplevel widget 65, 66
URL 63
working 78, 79
TNotebook widget 191
Toplevel widget 65, 66
TopLevel widget 16
TProgressbar widget 191
Treeview widget 192
TSeparator widget 192
TSizegrip widget 192
ttk widget 181, 182
custom style, creating 186-189
special widgets 190
style 183-185
theme 182, 183
themes, obtaining 189, 190
ttk widgets, special widgets
TCombobox widget 190
TNotebook widget 191
TProgressbar widget 191
Treeview widget 192
TSeparator widget 192

U

update_config_data function 239
update_label function 206
update_record function 252

V

variable attribute 23

W

WidgetDemo program 191
Widget toolbar 15
widget tree 15
winnative theme 183
write_full_record 254

Python GUI Programming with PAGE

DESCRIPTION

PAGE is a Tkinter-based GUI designer for Python, available for free and as an open-source tool. It generates native Python code, enabling users to swiftly create Graphical Interfaces for their Python programs. If you're eager to delve into GUI development for your Python programs, then this book is your go-to resource.

This comprehensive book is your guide from installing the PAGE designer to mastering the creation of complex GUI interfaces. It covers a wide range of topics, from building front-end interfaces for SQLite databases (and other databases) to utilizing the Canvas widget for drawing shapes and text. The book explores various aspects, including working with standard Tk widgets (such as buttons and entry), leveraging the capabilities of the ttk toolkit, and extending GUI functionality through third-party widget libraries and custom widgets. Each chapter presents real-world usable programs that challenge readers to enhance their skills and become more productive in your programming careers.

By the end of the book, you will possess the skills and knowledge to confidently develop your own GUI Python programs.

KEY FEATURES

- Explore RAD GUI programming concepts in Python with real-world examples.
- Explore exciting third-party libraries for enhanced GUI development.
- Learn to incorporate third-party Tk and ttk widgets into your own Python programs.

WHAT YOU WILL LEARN

- Learn how to install and start PAGE correctly.
- Explore the various widgets in the Tk and ttk toolkit that PAGE supports.
- Learn how to use graphic images in your projects.
- Understand how to communicate with a SQLite database and display data from it.
- Create projects that have more than one form and learn how to control those forms.

WHO THIS BOOK IS FOR

This book is for beginners and advanced Python programmers who wish to create attractive and logical user interfaces for Python. It is also for professionals who wish to explore Rapid Application Development (RAD) techniques for creating Python GUI programs.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-837-8