

Python GUI Programming

Cookbook

Third Edition

Develop functional and responsive user interfaces with tkinter and PyQt5



Burkhard Meier

Packt

www.packt.com

Python GUI Programming Cookbook

Third Edition

Develop functional and responsive user interfaces with tkinter
and PyQt5

Burkhard Meier

Packt

BIRMINGHAM - MUMBAI

Python GUI Programming Cookbook

Third Edition

Copyright © 2019 Packt Publishing

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 Packt Publishing 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.

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

Commissioning Editor: Richa Tripathi

Acquisition Editor: Denim Pinto

Content Development Editor: Pathikrit Roy

Senior Editor: Afshaan Khan

Technical Editor: Ketan Kamble

Copy Editor: Safis Editing

Project Coordinator: Prajakta Naik

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Deepika Naik

First published: December 2015

Second edition: May 2017

Third edition: October 2019

Production reference: 1111019

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83882-754-0

www.packt.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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 Packt books and eBooks.

Contributors

About the author

Burkhard Meier has more than 19 years of professional experience working in the software industry as a software tester and developer, specializing in software test automation development, execution, and analysis. He has a very strong background in Python 3 software test automation development, as well as in SQL relational database administration, the development of stored procedures, and debugging code.

His previous jobs include working as a senior test automation engineer and designer for InfoGenesis (now Agilysys), QAD, InTouch Health, and FLIR Systems.

Over the past three years, he has also developed several video courses on Python for Packt, the latest course being *Mastering Object-Oriented Programming with Python*.

I would like to thank all truly great artists, such as Leonardo da Vinci, Charles Baudelaire, Edgar Allan Poe, and so many others for bringing beauty into our human lives. This book is about creating very beautiful GUIs written in the Python programming language, and it was inspired by these truly great artists.

I would like to thank all of the great people that made this book possible. Without any of you, this book would only exist in my mind. I would like to especially thank all of my editors at Packt Publishing: Tanvi, Sonali, Anurag, Prashant, Vivek, Arwa, Sumeet, Saurabh, Pramod, Nikhil, Ketan, and so many others. I would also like to thank all of the reviewers of the code of this book. Without them, this book would be harder to read and apply to real-world problems.

Last but not least, I'd like to thank my wife, our daughter, and our parents for the emotional support they provided so successfully during the writing of the second and third editions of this book. I'd also like to give thanks to the creator of the very beautiful and powerful programming language that Python truly is. Thank you, Guido.

About the reviewers

Maurice HT Ling is a research assistant professor at the Perdana University School of Data Sciences. He obtained his BSc (Hons) in molecular and cell biology from the University of Melbourne, Australia, in 2004, and a BSc in computing from the University of Portsmouth, United Kingdom, in 2007, before obtaining his Ph.D. in bioinformatics from the University of Melbourne, Australia, in 2009.

He cofounded Python User Group (Singapore) and is instrumental in inaugurating PyCon Asia-Pacific as one of the three major Python conferences worldwide. In his free time, he likes to read, enjoy a cup of coffee, write in his personal journal, and philosophize on various aspects of life.

Rahul Shendge has a bachelor's degree in computer engineering from the University of Pune and is certified in multiple technologies. He is an open source enthusiast and works as a senior software engineer. He has worked in multiple domains, including finance, healthcare, and education.

He has hands-on experience in the cloud, and in designing trading algorithms with machine learning. He is constantly exploring technical novelties and is open-minded and eager to learn about new technologies. He is passionate about helping clients make valuable business decisions using analytics in their respective areas. His main interests are to work on and explore data analytics solutions.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Table of Contents

Preface	1
Chapter 1: Creating the GUI Form and Adding Widgets	8
Creating our first Python GUI	10
Getting ready	10
How to do it...	10
How it works...	12
Preventing the GUI from being resized	13
Getting ready	14
How to do it...	14
How it works...	15
Adding a label to the GUI form	16
Getting ready	16
How to do it...	16
How it works...	17
Creating buttons and changing their text attributes	18
Getting ready	18
How to do it...	19
How it works...	20
Creating textbox widgets	21
Getting ready	21
How to do it...	21
How it works...	22
Setting the focus to a widget and disabling widgets	24
Getting ready	24
How to do it...	24
How it works...	26
Creating combobox widgets	27
Getting ready	27
How to do it...	27
How it works...	29
There's more...	30
Creating a check button with different initial states	31
Getting ready	31
How to do it...	32
How it works...	34
Using radio button widgets	34
Getting ready	35
How to do it...	35
How it works...	37

There's more...	38
Using scrolled text widgets	38
Getting ready	38
How to do it...	38
How it works...	40
Adding several widgets in a loop	41
Getting ready	41
How to do it...	41
How it works...	43
Chapter 2: Layout Management	44
Arranging several labels within a label frame widget	45
Getting ready	46
How to do it...	46
How it works...	49
See also...	49
Using padding to add space around widgets	49
Getting ready	50
How to do it...	50
How it works...	53
Dynamically expanding the GUI using widgets	54
Getting ready	54
How to do it...	56
How it works...	59
There's more...	59
Aligning GUI widgets by embedding frames within frames	59
Getting ready	59
How to do it...	60
How it works...	63
Creating menu bars	64
Getting ready	64
How to do it...	65
How it works...	71
There's more...	72
Creating tabbed widgets	72
Getting ready	72
How to do it...	72
How it works...	79
Using the grid layout manager	80
Getting ready...	80
How to do it...	80
How it works...	81
Chapter 3: Look and Feel Customization	83
Creating message boxes – information, warning, and error	84
Getting ready	84

How to do it...	85
How it works...	89
How to create independent message boxes	89
Getting ready	90
How to do it...	90
How it works...	93
How to create the title of a tkinter window form	94
Getting ready	94
How to do it...	94
How it works...	95
Changing the icon of the main root window	95
Getting ready	95
How to do it...	95
How it works...	96
Using a spin box control	97
Getting ready	97
How to do it...	97
How it works...	101
Applying relief – the sunken and raised appearance of widgets	102
Getting ready	102
How to do it...	102
How it works...	104
Creating tooltips using Python	105
Getting ready	105
How to do it...	106
How it works...	108
Adding Progressbar to the GUI	109
Getting ready	109
How to do it...	109
How it works...	112
How to use the canvas widget	113
Getting ready	113
How to do it...	114
How it works...	115
Chapter 4: Data and Classes	116
How to use StringVar()	117
Getting ready	117
How to do it...	118
How it works...	123
How to get data from a widget	124
Getting ready	124
How to do it...	124
How it works...	126
Using module-level global variables	126

Getting ready	126
How to do it...	127
How it works...	131
How coding in classes can improve the GUI	132
Getting ready	132
How to do it...	132
How it works...	134
Writing callback functions	137
Getting ready	138
How to do it...	138
How it works...	139
Creating reusable GUI components	139
Getting ready	139
How to do it...	140
How it works...	141
Chapter 5: Matplotlib Charts	142
Installing Matplotlib using pip with the .whl extension	144
Getting ready	144
How to do it...	147
How it works...	150
Creating our first chart	151
Getting ready	151
How to do it...	151
How it works...	152
There's more...	153
Placing labels on charts	153
Getting ready	153
How to do it...	154
How it works...	158
How to give the chart a legend	159
Getting ready	159
How to do it...	159
How it works...	163
Scaling charts	164
Getting ready	164
How to do it...	165
How it works...	166
Adjusting the scale of charts dynamically	167
Getting ready	167
How to do it...	167
How it works...	169
Chapter 6: Threads and Networking	172
How to create multiple threads	174
Getting ready	174

How to do it...	174
How it works...	177
Starting a thread	178
Getting ready	178
How to do it...	178
How it works...	182
Stopping a thread	183
Getting ready	183
How to do it...	183
How it works...	185
How to use queues	186
Getting ready	187
How to do it...	187
How it works...	192
Passing queues among different modules	193
Getting ready	194
How to do it...	194
How it works...	196
Using dialog widgets to copy files to your network	198
Getting ready	198
How to do it...	198
How it works...	206
Using TCP/IP to communicate via networks	208
Getting ready	209
How to do it...	209
How it works...	210
Using urlopen to read data from websites	212
Getting ready	212
How to do it...	212
How it works...	215
Chapter 7: Storing Data in Our MySQL Database via Our GUI	217
Installing and connecting to a MySQL server from Python	219
Getting ready	219
How to do it...	219
How it works...	223
Configuring the MySQL database connection	225
Getting ready	225
How to do it...	225
How it works...	228
Designing the Python GUI database	230
Getting ready	230
How to do it...	230
How it works...	235
Using the SQL INSERT command	239

Getting ready	239
How to do it...	240
How it works...	242
Using the SQL UPDATE command	242
Getting ready	242
How to do it...	243
How it works...	247
Using the SQL DELETE command	247
Getting ready	248
How to do it...	248
How it works...	251
Storing and retrieving data from our MySQL database	252
Getting ready	253
How to do it...	253
How it works...	256
Using MySQL Workbench	257
Getting ready	257
How to do it...	257
How it works...	262
There's more...	263
Chapter 8: Internationalization and Testing	264
Displaying widget text in different languages	265
Getting ready	266
How to do it...	266
How it works...	268
Changing the entire GUI language all at once	268
Getting ready	268
How to do it...	269
How it works...	272
Localizing the GUI	272
Getting ready	273
How to do it...	273
How it works...	276
Preparing the GUI for internationalization	276
Getting ready	277
How to do it...	277
How it works...	280
How to design a GUI in an agile fashion	281
Getting ready	282
How to do it...	282
How it works...	284
Do we need to test the GUI code?	284
Getting ready	285
How to do it...	285

How it works...	288
Setting debug watches	289
Getting ready	289
How to do it...	289
How it works...	291
Configuring different debug output levels	292
Getting ready	293
How to do it...	293
How it works...	295
Creating self-testing code using Python's <code>__main__</code> section	296
Getting ready	297
How to do it...	297
How it works...	300
Creating robust GUIs using unit tests	301
Getting ready	302
How to do it...	302
How it works...	304
How to write unit tests using the Eclipse PyDev IDE	306
Getting ready	307
How to do it...	307
How it works...	310
Chapter 9: Extending Our GUI with the wxPython Library	313
Installing the wxPython library	314
Getting ready	315
How to do it...	315
How it works...	316
Creating our GUI in wxPython	317
Getting ready	317
How to do it...	318
How it works...	321
Quickly adding controls using wxPython	321
Getting ready	322
How to do it...	322
How it works...	326
Trying to embed a main wxPython app in a main tkinter app	327
Getting ready	328
How to do it...	328
How it works...	330
Trying to embed our tkinter GUI code into wxPython	331
Getting ready	331
How to do it...	331
How it works...	333
Using Python to control two different GUI frameworks	334
Getting ready	334

How to do it...	334
How it works...	338
Communicating between the two connected GUIs	339
Getting ready	339
How to do it...	339
How it works...	342
Chapter 10: Building GUIs with PyQt5	345
Installing PyQt5	347
Getting ready	347
How to do it...	347
How it works...	348
Installing the PyQt5 Designer tool	349
Getting ready	349
How to do it...	349
How it works...	350
Writing our first PyQt5 GUI	351
Getting ready	351
How to do it...	351
How it works...	352
Changing the title of the GUI	352
Getting ready	352
How to do it...	353
How it works...	353
There's more...	354
Refactoring our code into object-oriented programming	354
Getting ready	354
How to do it...	355
How it works...	355
Inheriting from QMainWindow	356
Getting ready	356
How to do it...	356
How it works...	357
Adding a status bar widget	357
Getting ready	357
How to do it...	357
How it works...	358
Adding a menu bar widget	358
Getting ready	359
How to do it...	359
How it works...	360
Starting the PyQt5 Designer tool	360
Getting ready	360
How to do it...	360
How it works...	362

Previewing the form within the PyQt5 Designer	363
Getting ready	364
How to do it...	364
How it works...	365
Saving the PyQt5 Designer form	366
Getting ready	366
How to do it...	366
How it works...	368
Converting Designer .ui code into .py code	368
Getting ready	368
How to do it...	369
How it works...	370
Understanding the converted Designer code	371
Getting ready	371
How to do it...	371
How it works...	373
Building a modular GUI design	374
Getting ready	374
How to do it...	374
How it works...	375
Adding another menu item to our menu bar	376
Getting ready	376
How to do it...	376
How it works...	379
There's more...	379
Connecting functionality to the Exit menu item	379
Getting ready	380
How to do it...	380
How it works...	381
Adding a Tab Widget via the Designer	383
Getting ready	383
How to do it...	384
How it works...	385
Using layouts in the Designer	386
Getting ready	386
How to do it...	387
How it works...	387
Adding buttons and labels in the Designer	388
Getting ready	388
How to do it...	389
How it works...	393
There's more...	395
Chapter 11: Best Practices	398
Avoiding spaghetti code	399

Getting ready	399
How to do it...	399
How it works...	403
Using <code>__init__</code> to connect modules	406
Getting ready	406
How to do it...	406
How it works...	410
Mixing fall-down and OOP coding	412
Getting ready	412
How to do it...	412
How it works...	416
Using a code naming convention	416
Getting ready	416
How to do it...	417
How it works...	418
There's more...	418
When not to use OOP	420
Getting ready	420
How to do it...	421
How it works...	424
How to use design patterns successfully	425
Getting ready	425
How to do it...	425
How it works...	427
Avoiding complexity	428
Getting ready	428
How to do it...	428
How it works...	436
GUI design using multiple notebooks	437
Getting ready	438
How to do it...	438
How it works...	445
Other Books You May Enjoy	452
Index	455

Preface

In the third edition of this book, we will explore the beautiful world of **graphical user interfaces (GUIs)** using the Python programming language. We will be using the latest version of Python 3. All of the recipes from the first and second editions are included in this edition, except for the outdated OpenGL library, which is not very Pythonic, after all. We have added an entirely new chapter to the third edition, and we have dramatically changed the style of this third edition to give it more of a cookbook format. By doing so, hopefully, it is easier to apply the recipes to real-world programming situations, providing tested and working solutions.

This is a programming cookbook. Every chapter is self-contained and explains a certain programming solution. We will start very simply, yet throughout the course of this book, we will build a working application written in Python 3. Each recipe will extend the building of this application. Along the way, we will talk about networks, queues, databases, the PyQt5 graphical library, and many more technologies. We will apply design patterns and use best practices.

The book assumes that you have some experience of using the Python programming language, but that is not really required to successfully use this book. This book can also be used as an introduction to the Python programming language, if, and only if, you are dedicated in your desire to become a Pythonic programmer.

If you are an experienced developer in any other language, you will have a fun time extending your professional toolbox by adding the ability to write GUIs using Python to your toolbox.

Who this book is for

This book is for programmers who wish to create a GUI. You might be surprised by what we can achieve by creating beautiful, functional, and powerful GUIs using the Python programming language. Python is a wonderful, intuitive programming language, and is very easy to learn.

I invite you to start on this journey now. It will be a lot of fun!

What this book covers

Chapter 1, *Creating the GUI Form and Adding Widgets*, explains how to develop our first GUI in Python. We will start with the minimum code required to build a running GUI application. Each recipe then adds different widgets to the GUI form.

Chapter 2, *Layout Management*, explores how to arrange widgets to create our Python GUI. The grid layout manager is one of the most important layout tools built into `tkinter` that we will be using.

Chapter 3, *Look and Feel Customization*, offers several examples of how to create a GUI with good look and feel. On a practical level, we will add functionality to the **Help | About** menu item that we created in one of the recipes.

Chapter 4, *Data and Classes*, discusses saving the data our GUI displays. We will start using **object-oriented programming (OOP)** in order to extend Python's built-in functionality.

Chapter 5, *Matplotlib Charts*, explains how to create beautiful charts that visually represent data. Depending on the format of the data source, we can plot one or several columns of data within the same chart.

Chapter 6, *Threads and Networking*, explains how to extend the functionality of our Python GUI using threads, queues, and network connections. This will show us that our GUI is not limited at all to the local scope of our PC.

Chapter 7, *Storing Data in Our MySQL Database via Our GUI*, shows us how to connect to a MySQL database server. The first recipe in this chapter will show you how to install the free MySQL Server Community Edition, while, in the following recipes, we will create databases and tables, and then load data into those tables and modify it. We will also read the data back out from the MySQL server into our GUI.

Chapter 8, *Internationalization and Testing*, explains how to internationalize our GUI by displaying text on labels, buttons, tabs, and other widgets in different languages. We will start with a simple example and then explore how we can prepare our GUI for internationalization at the design level. We will also explore several ways to automatically test our GUI using Python's built-in unit testing framework.

Chapter 9, *Extending Our GUI with the wxPython Library*, introduces another Python GUI toolkit that currently does not ship with Python. It is called **wxPython**, and we will be using the Phoenix version of wxPython, which was designed to work well with Python 3.

Chapter 10, *Building GUIs with PyQt5*, shows you how to use the wonderful PyQt5 GUI programming framework. Tesla Motors uses this to build their GUI software, and, in this chapter, we will explore the beautiful world of drag and drop IDE GUI development using Python binding with Qt5, which, underneath the hood, is built upon C++. If you wish to get serious about Python GUI development, you need to study this chapter in addition to `tkinter`.

Chapter 11, *Best Practices*, explores different best practices that can help us to build our GUI in an efficient way and keep it both maintainable and extendable. Best practices are applicable to any good code, and our GUI is no exception when it comes to designing and implementing good software practices.

To get the most out of this book

To make optimum use of the content in this book, please bear the following points in mind:

- All the recipes in this book were developed using Python 3.7 on a Windows 10 64-bit OS. They have not been tested on any other configuration. As Python is a cross-platform language, the code from each recipe is expected to run everywhere.
- If you are using a Mac, it does come with built-in Python, yet it might be missing some modules such as `tkinter`, which we will use throughout this book.
- We are using Python 3.7, and the creator of Python intentionally chose not to make it backward-compatible with Python 2. If you are using a Mac or Python 2, you might have to install Python 3.7 from www.python.org in order to successfully run the recipes in this book.
- If you really wish to run the code in this book on Python 2.7, you will have to make some adjustments. For example, `tkinter` in Python 2.x has an uppercase T. The Python 2.7 print statement is a function in Python 3.7 and requires parentheses.
- While the **End of Life (EOL)** for the Python 2.x branch has been extended to the year 2020, I would strongly recommend that you start using Python 3.7 and later.
- Why hold on to the past, unless you really have to? Here is a link to the **Python Enhancement Proposal (PEP) 373** that refers to the EOL of Python 2: <https://www.python.org/dev/peps/pep-0373/>.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781838827540_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Here is an overview of the Python modules (ending in a .py extension) for this chapter".

A block of code is set as follows:

```
action = ttk.Button(win, text="Click Me!", command=click_me)
action.grid(column=2, row=1)
```

Any command-line input or output is written as follows:

```
pip install pyqt5
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on the **File** menu and then click on **New**."

Warnings or important notes appear like this.



Tips and tricks appear like this.



Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 copyright@packt.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 authors.packtpub.com.

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 Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Creating the GUI Form and Adding Widgets

In this chapter, we will develop our first GUI in Python. We will start with the minimum code required to build a running GUI application. Each recipe then adds different widgets to the GUI form.

We will start by using the `tkinter` GUI toolkit.



`tkinter` ships with Python. There is no need to install it once you have installed Python version 3.7 or later. The `tkinter` GUI toolkit enables us to write GUIs with Python.

The old world of the DOS Command Prompt has long been outdated. Some developers still like it for development work. The end user of your program expects a more modern, good-looking GUI.

In this book, you will learn how to develop GUIs using the Python programming language.



By starting with the minimum amount of code, we can see the **pattern** every GUI written with `tkinter` and Python follows. First come the `import` statements, followed by the creation of a `tkinter` class. We then can call methods and change attributes. At the end, we always call the Windows event loop. Now we can run the code.

We progress from the most simple code, adding more and more functionality with each following recipe, introducing different widget controls and how to change and retrieve attributes.

In the first two recipes, we will show the entire code, consisting of only a few lines of code. In the following recipes, we will only show the code to be added to the previous recipes because, otherwise, the book would get too long, and seeing the same code over and over again is rather boring.

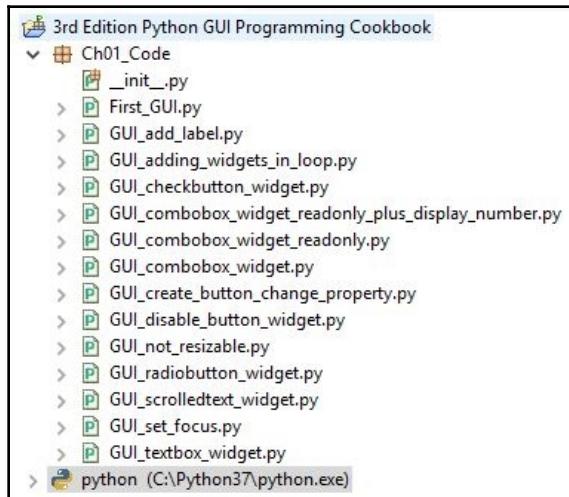


If you don't have the time to type the code yourself, you can download all of the code for the entire book from <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition>.

At the beginning of each chapter, I will show the Python modules that belong to each chapter. I will then reference the different modules that belong to the code shown, studied, and run.

By the end of this chapter, we will have created a working GUI application that consists of labels, buttons, textboxes, comboboxes, check buttons in various states, and radio buttons that change the background color of the GUI.

Here is an overview of the Python modules (ending in a .py extension) for this chapter:



In this chapter, we start creating amazing GUIs using Python 3.7 or later. We will cover the following topics:

- Creating our first Python GUI
- Preventing the GUI from being resized
- Adding a label to the GUI form

- Creating buttons and changing their text attributes
- Creating textbox widgets
- Setting the focus to a widget and disabling widgets
- Creating combobox widgets
- Creating a check button with different initial states
- Using radio button widgets
- Using scrolled text widgets
- Adding several widgets in a loop

Creating our first Python GUI

Python is a very powerful programming language. It ships with the built-in `tkinter` module. In only a few lines of code (four, to be precise) we can build our first Python GUI.



`tkinter` is a Python **interface** to `tk`. `tk` is a GUI toolkit and related to `Tcl`, which is a tool command language. You can learn more about `tk` at <https://docs.python.org/3/library/tk.html>.

Another website related to `tcl` and `tk` is <https://www.tcl.tk/>.

Getting ready

To follow this recipe, a working Python development environment is a prerequisite. The IDLE GUI, which ships with Python, is enough to start. IDLE was built using `tkinter`!

How to do it...

Let's take a look at how to create our first Python GUI:

1. Create a new Python module and name it `First_GUI.py`.
2. At the top of the `First_GUI.py` module, import `tkinter`:

```
import tkinter as tk
```

3. Create an instance of the Tk class:

```
win = tk.Tk()
```

4. Use the instance variable to set a title:

```
win.title("Python GUI")
```

5. Start the window's main event loop:

```
win.mainloop()
```

The following screenshot shows the four lines of `First_GUI.py` required to create the resulting GUI:

```
6#=====
7# imports
8#=====
9import tkinter as tk
10
11# Create instance
12win = tk.Tk()
13
14# Add a title
15win.title("Python GUI")
16
17#=====
18# Start GUI
19#=====
20win.mainloop()
```

6. Run the GUI module. On executing the preceding code, the following output is obtained:



Now, let's go behind the scenes to understand the code better.

How it works...

In *line 9*, we import the built-in `tkinter` module and alias it as `tk` to simplify our Python code. In *line 12*, we create an instance of the `Tk` class by calling its constructor (the parentheses appended to `Tk` turns the class into an instance). We are using the `tk` alias so we don't have to use the longer word `tkinter`. We are assigning the class instance to a variable named `win` (short for a window) so that we can access the class attributes via this variable. As Python is a dynamically typed language, we did not have to declare this variable before assigning to it, and we did not have to give it a specific type. Python *infers* the type from the assignment of this statement. Python is a strongly typed language, so every variable always has a type. We just don't have to specify its type beforehand like in other languages. This makes Python a very powerful and productive language to program in.

A little note about classes and types: In Python, every variable always has a type. We cannot create a variable that does not have a type. Yet, in Python, we do not have to declare the type beforehand, as we have to do in the C programming language.

Python is smart enough to infer the type. C#, at the time of writing this book, also has this capability.

Using Python, we can create our own classes using the `class` keyword instead of the `def` keyword.



In order to assign the class to a variable, we first have to create an instance of our class. We create the instance and assign this instance to our variable, for example:

```
class AClass(object):
    print('Hello from AClass')
    class_instance = AClass()
```

Now, the `class_instance` variable is of the `AClass` type. If this sounds confusing, do not worry. We will cover **object-oriented programming (OOP)** in the coming chapters.

In *line 15*, we use the instance variable (`win`) of the class to give our window a title by calling the `title()` method, passing in a string.



You might have to enlarge the running GUI to see the entire title.

In line 20, we start the window's event loop by calling the `mainloop` method on the class instance, `win`. Up to this point in our code, we have created an instance and set one attribute (the window title), but the GUI will not be displayed until we start the main event loop.

An event loop is a mechanism that makes our GUI work. We can think of it as an endless loop where our GUI is waiting for events to be sent to it. A button click creates an event within our GUI, or our GUI being resized also creates an event.



We can write all of our GUI code in advance and nothing will be displayed on the user's screen until we call this endless loop (`win.mainloop()` in the preceding code). The event loop ends when the user clicks the red X button or a widget that we have programmed to end our GUI. When the event loop ends, our GUI also ends.

This recipe used the minimum amount of Python code to create our first GUI program. However, throughout this book we will use OOP when it makes sense.

We've successfully learned how to create our first Python GUI. Now, let's move on to the next recipe.

Preventing the GUI from being resized

By default, a GUI created using `tkinter` can be resized. This is not always ideal. The widgets we place onto our GUI forms might end up being resized in an improper way, so in this recipe, we will learn how to prevent our GUI from being resized by the user of our GUI application.

Getting ready

This recipe extends the previous one, *Creating our first Python GUI*, so one requirement is to have typed the first recipe yourself into a project of your own. Alternatively, you can download the code from <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition/>.

How to do it...

Here are the steps to prevent the GUI from being resized:

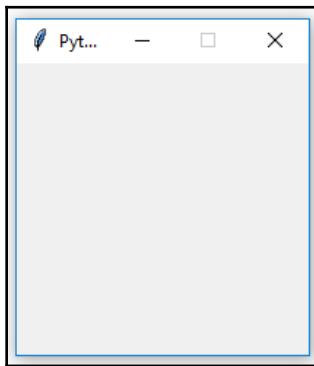
1. Start with the module from the previous recipe and save it as `Gui_not_resizable.py`.
2. Use the `Tk` instance variable, `win`, to call the `resizable` method:

```
win.resizable(False, False)
```

Here is the code to prevent the GUI from being resized (`GUI_not_resizable.py`):

```
6#=====
7# imports
8#####
9import tkinter as tk
10
11# Create instance
12win = tk.Tk()
13
14# Add a title
15win.title("Python GUI")
16
17# Disable resizing the GUI by passing in False/False
18win.resizable(False, False)
19
20# Enable resizing x-dimension, disable y-dimension
21# win.resizable(True, False)
22
23#####
24# Start GUI
25#####
26win.mainloop()
```

3. Run the code. Running the code creates this GUI:



Let's go behind the scenes to understand the code better.

How it works...

Line 18 prevents the Python GUI from being resized.

The `resizable()` method is of the `Tk()` class and, by passing in `(False, False)`, we prevent the GUI from being resized. We can disable both the *x* and *y* dimensions of the GUI from being resized, or we can enable one or both dimensions by passing in `True` or any number other than zero. `(True, False)` would enable the *x* dimension but prevent the *y* dimension from being resized.

Running this code will result in a GUI similar to the one we created in the first recipe. However, the user can no longer resize it. Also, note how the maximize button in the toolbar of the window is grayed out.

Why is this important? Because once we add widgets to our form, resizing our GUI can make it not look the way we want it to look. We will add widgets to our GUI in the next recipes, starting with *Adding a label to the GUI form*.

We also added comments to our code in preparation for the recipes contained in this book.



In visual programming IDEs such as Visual Studio .NET, C# programmers often do not think of preventing the user from resizing the GUI they developed in this language. This creates inferior GUIs. Adding this one line of Python code can make our users appreciate our GUI.

We've successfully learned how to prevent the GUI from being resized. Now, let's move on to the next recipe.

Adding a label to the GUI form

A label is a very simple widget that adds value to our GUI. It explains the purpose of the other widgets, providing additional information. This can guide the user to the meaning of an `Entry` widget, and it can also explain the data displayed by widgets without the user having to enter data into it.

Getting ready

We are extending the first recipe, *Creating our first Python GUI*. We will leave the GUI resizable, so don't use the code from the second recipe (or comment the `win.resizable` line out).

How to do it...

Perform the following steps to add a label to the GUI from:

1. Start with the `First_GUI.py` module and save it as `GUI_add_label.py`.
2. Import `ttk`:

```
from tkinter import ttk
```

3. Use `ttk` to add a label:

```
ttk.Label(win, text="A Label")
```

4. Use the grid layout manager to position the label:

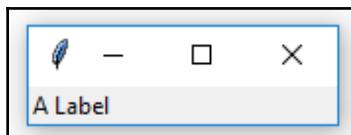
```
.grid(column=0, row=0)
```

In order to add a `Label` widget to our GUI, we will import the `ttk` module from `tkinter`. Please note the two `import` statements on lines 9 and 10.

The following code is added just above `win.mainloop()`, which is located at the bottom of the first and second recipes (`GUI_add_label.py`):

```
6#=====
7# imports
8#=====
9import tkinter as tk
10from tkinter import ttk
11
12# Create instance
13win = tk.Tk()
14
15# Add a title
16win.title("Python GUI")
17
18# Adding a Label
19ttk.Label(win, text="A Label").grid(column=0, row=0)
20
21#=====
22# Start GUI
23#=====
24win.mainloop()
```

5. Run the code and observe how a label is added to our GUI:



Let's go behind the scenes to understand the code better.

How it works...

In line 10 of the preceding code, we import a separate module from the `tkinter` package. The `ttk` module has some advanced widgets such as a notebook, progress bar, labels, and buttons that look different. These help to make our GUI look better. In a sense, `ttk` is an extension within the `tkinter` package.

We still need to import the `tkinter` package, but we need to specify that we now want to also use `ttk` from the `tkinter` package.



`ttk` stands for *themed tk*. It improves our GUI's look and feel. You can find more information at <https://docs.python.org/3/library/tkinter.ttk.html>.

Line 19 adds the label to the GUI, just before we call `mainloop`.

We pass our window instance into the `ttk.Label` constructor and set the `text` attribute. This becomes the text our `Label` will display. We also make use of the grid layout manager, which we'll explore in much more depth in Chapter 2, *Layout Management*.

Observe how our GUI suddenly got much smaller than in the previous recipes. The reason why it became so small is that we added a widget to our form. Without a widget, the `tkinter` package uses a default size. Adding a widget causes optimization, which generally means using as little space as necessary to display the widget(s). If we make the text of the label longer, the GUI will expand automatically. We will cover this automatic form size adjustment in a later recipe in Chapter 2, *Layout Management*.

Try resizing and maximizing this GUI with a label and watch what happens. We've successfully learned how to add a label to the GUI form.

Now, let's move on to the next recipe.

Creating buttons and changing their text attributes

In this recipe, we will add a button widget, and we will use this button to change an attribute of another widget that is a part of our GUI. This introduces us to callback functions and event handling in a Python GUI environment.

Getting ready

This recipe extends the previous one, *Adding a label to the GUI form*. You can download the entire code from <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition/>.

How to do it...

In this recipe, we will update the label we added in the previous recipe as well as the `text` attribute of the button. The steps to add a button that performs an action when clicked are as follows:

1. Start with the `GUI_add_label.py` module and save it as `GUI_create_button_change_property.py`.
2. Define a function and name it `click_me()`:

```
def click_me():
```

3. Use `ttk` to create a button and give it a `text` attribute:

```
action.configure(text="** I have been Clicked! **")
a_label.configure(foreground='red')
a_label.configure(text='A Red Label')
```

4. Bind the function to the button:

```
action = ttk.Button(win, text="Click Me!", command=click_me)
```

5. Use the grid layout to position the button:

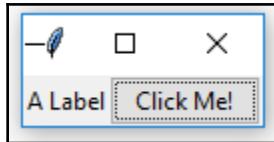
```
action.grid(column=1, row=0)
```

The preceding instructions produce the following code (`GUI_create_button_change_property.py`):

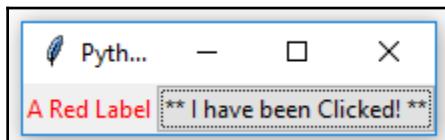
```
18 # Adding a Label that will get modified
19 a_label = ttk.Label(win, text="A Label")
20 a_label.grid(column=0, row=0)
21
22 # Button Click Event Function
23 def click_me():
24     action.configure(text="** I have been Clicked! **")
25     a_label.configure(foreground='red')
26     a_label.configure(text='A Red Label')
27
28 # Adding a Button
29 action = ttk.Button(win, text="Click Me!", command=click_me)
30 action.grid(column=1, row=0)
31
32 =====
33 # Start GUI
34 =====
35 win.mainloop()
```

6. Run the code and observe the output.

The following screenshot shows how our GUI looks before clicking the button:



After clicking the button, the color of the label changed and so did the text of the button, which can be seen in the following screenshot:



Let's go behind the scenes to understand the code better.

How it works...

In line 19, we assign the label to a variable, `a_label`, and in line 20, we use this variable to position the label within the form. We need this variable in order to change its attributes in the `click_me()` function. By default, this is a module-level variable, so as long as we declare the variable above the function that calls it, we can access it inside the function.

Line 23 is the event handler that is invoked once the button gets clicked.

In line 29, we create the button and bind the command to the `click_me()` function.



GUIs are event-driven. Clicking the button creates an event. We bind what happens when this event occurs in the callback function using the `command` attribute of the `ttk.Button` widget. Notice how we do not use parentheses, only the name `click_me`.

Lines 20 and 30 both use the grid layout manager, which will be discussed in [Chapter 2, Layout Management](#), in the *Using the grid layout manager* recipe. This aligns both the label and the button. We also change the text of the label to include the word `red` to make it more obvious that the color has been changed. We will continue to add more and more widgets to our GUI, and we will make use of many built-in attributes in the other recipes of this book.

We've successfully learned how to create buttons and change their text attributes. Now, let's move on to the next recipe.

Creating textbox widgets

In `tkinter`, a typical one-line textbox widget is called `Entry`. In this recipe, we will add such an `Entry` widget to our GUI. We will make our label more useful by describing what the `Entry` widget is doing for the user.

Getting ready

This recipe builds upon the *Creating buttons and changing their text attributes* recipe, so download it from the repository and start working on it.

How to do it...

Follow these steps to create textbox widgets:

1. Start with the `GUI_create_button_change_property.py` module and save it as `GUI_textbox_widget.py`.
2. Use the `tk` alias of `tkinter` to create a `StringVar` variable:

```
name = tk.StringVar()
```

3. Create a `ttk.Entry` widget and assign it to another variable:

```
name_entered = ttk.Entry(win, width=12, textvariable=name)
```

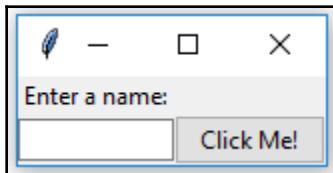
4. Use this variable to position the `Entry` widget:

```
name_entered.grid(column=0, row=1)
```

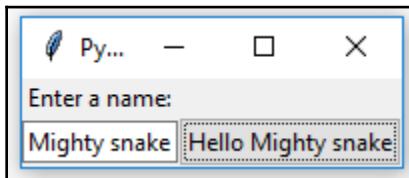
The preceding instructions produce the following code
(GUI_textbox_widget.py):

```
22 # Modified Button Click Function
23 def click_me():
24     action.configure(text='Hello ' + name.get())
25
26 # Changing our Label
27 ttk.Label(win, text="Enter a name:").grid(column=0, row=0)
28
29 # Adding a Text box Entry widget
30 name = tk.StringVar()
31 name_entered = ttk.Entry(win, width=12, textvariable=name)
32 name_entered.grid(column=0, row=1)
```

5. Run the code and observe the output; our GUI looks like this:



6. Enter some text and click the button; we will see that there is a change in the GUI, which is as follows:



Let's go behind the scenes to understand the code better.

How it works...

In *step 1* we are creating a new Python module, and in *step 2* we are adding a `StringVar` type of `tkinter` and saving it in the `name` variable. We use this variable when we are creating an `Entry` widget and assigning it to the `textvariable` attribute of the `Entry` widget. Whenever we type some text into the `Entry` widget, this text will be saved in the `name` variable.

In step 4, we position the Entry widget and the preceding screenshot shows the entire code.

In line 24, as shown in the screenshot, we get the value of the Entry widget using `name.get()`.

When we created our button, we saved a reference to it in the `action` variable. We use the `action` variable to call the `configure` method of the button, which then updates the text of our button.



We have not used OOP yet, so how come we can access the value of a variable that was not even declared yet? Without using OOP classes, in Python procedural coding, we have to physically place a name above a statement that tries to use that name. So, how does this work (it does)? The answer to this is that the button click event is a callback function, and by the time the button is clicked by a user, the variables referenced in this function are known and do exist.

Line 27 gives our label a more meaningful name; for now, it describes the textbox below it. We moved the button down next to the label to visually associate the two. We are still using the grid layout manager, which will be explained in more detail in [Chapter 2, Layout Management](#).

Line 30 creates a variable, `name`. This variable is bound to the Entry widget and, in our `click_me()` function, we are able to retrieve the value of the Entry widget by calling `get()` on this variable. This works like a charm.

Now we observe that while the button displays the entire text we entered (and more), the textbox Entry widget did not expand. The reason for this is that we hardcoded it to a width of 12 in line 31.



Python is a dynamically typed language and infers the type from the assignment. What this means is that if we assign a string to the `name` variable, it will be of the `string` type, and if we assign an integer to `name`, its type will be an integer.

Using `tkinter`, we have to declare the `name` variable as the `tk.StringVar()` type before we can use it successfully. The reason is that `tkinter` is not Python. We can use it with Python, but it is not the same language. See <https://wiki.python.org/moin/TkInter> for more information.

We've successfully learned how to create textbox widgets. Now, let's move on to the next recipe.

Setting the focus to a widget and disabling widgets

While our GUI is nicely improving, it would be more convenient and useful to have the cursor appear in the Entry widget as soon as the GUI appears.

In this recipe, we learn how to make the cursor appear in the Entry box for immediate text Entry rather than the need for the user to *click* into the Entry widget to give it the `focus` method before typing into the entry widget.

Getting ready

This recipe extends the previous recipe, *Creating textbox widgets*. Python is truly great. All we have to do to set the focus to a specific control when the GUI appears is call the `focus()` method on an instance of a `tkinter` widget we previously created. In our current GUI example, we assigned the `ttk.Entry` class instance to a variable named `name_entered`. Now, we can give it the focus.

How to do it...

Place the following code just above the previous code, which is located at the bottom of the module, and which starts the main window's event loop, like we did in the previous recipes:

1. Start with the `GUI_textbox_widget.py` module and save it as `GUI_set_focus.py`.
2. Use the `name_entered` variable we assigned the `ttk Entry` widget instance to and call the `focus()` method on this variable:

```
name_entered.focus()
```

The preceding instructions produce the following code (`GUI_set_focus.py`):

```
29 # Adding a Textbox Entry widget
30 name = tk.StringVar()
31 name_entered = ttk.Entry(win, width=12, textvariable=name)
32 name_entered.grid(column=0, row=1)
33
34 # Adding a Button
35 action = ttk.Button(win, text="Click Me!", command=click_me)
36 action.grid(column=1, row=1)
37
38 name_entered.focus()      # Place cursor into name Entry
39 #=====
40 # Start GUI
41 #=====
42 win.mainloop()
```

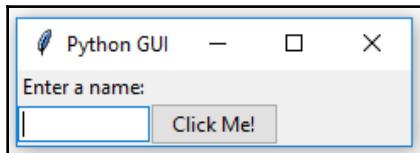
3. Run the code and observe the output.

If you get some errors, make sure you are placing calls to variables below the code where they are declared. We are not using OOP as of now, so this is still necessary. Later, it will no longer be necessary to do this.



On a Mac, you might have to set the focus to the GUI window first before being able to set the focus to the Entry widget in this window.

Adding line 38 of the Python code places the cursor in our text Entry widget, giving the text Entry widget the focus. As soon as the GUI appears, we can type into this textbox without having to click it first. The resulting GUI now looks like this, with the cursor inside the Entry widget:



Note how the cursor now defaults to residing inside the text entry box.

We can also disable widgets. Here, we are disabling the button to show the principle. In larger GUI applications, the ability to disable widgets gives you control when you want to make things read only. Most likely, those would be combobox widgets and Entry widgets, but as we have not yet gotten to those widgets yet, we will use our button.

To disable widgets, we will set an attribute on the widget. We can make the button disabled by adding the following code below line 37 of the Python code to create the button:

1. Use the `GUI_set_focus.py` module and save it as `GUI_disable_button_widget.py`.
2. Use the `action` button variable to call the `configure` method and set the `state` attribute to `disabled`:

```
action.configure(state='disabled')
```

3. Call the `focus()` method on the `name_entered` variable:

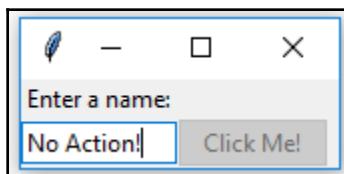
```
name_entered.focus()
```

The preceding instructions produce the following code

(`GUI_disable_button_widget.py`):

```
34 # Adding a Button
35 action = ttk.Button(win, text="Click Me!", command=click_me)
36 action.grid(column=1, row=1)
37 action.configure(state='disabled')      # Disable the Button Widget
38
39 name_entered.focus()      # Place cursor into name Entry
```

4. Run the code. After adding the preceding line of Python code, clicking the button no longer creates an action:



Let's go behind the scenes to understand the code better.

How it works...

This code is self-explanatory. In line 39, we set the focus to one control, and in line 37, we disable another widget. Good naming in programming languages helps to eliminate lengthy explanations. Later in this book, there will be some advanced tips on how to do this while programming at work or practicing our programming skills at home.

We've successfully learned how to set the focus to a widget and disable widgets. Now, let's move on to the next recipe.

Creating combobox widgets

In this recipe, we will improve our GUI by adding drop-down comboboxes that can have initial default values. While we can restrict the user to only certain choices, we can also allow the user to type in whatever they wish.

Getting ready

This recipe extends the previous recipe, *Setting the focus to a widget and disabling widgets*.

How to do it...

We insert another column between the Entry widget and the Button widget using the grid layout manager. Here is the Python code:

1. Start with the `GUI_set_focus.py` module and save it as `GUI_combobox_widget.py`.
2. Change the button column to 2:

```
action = ttk.Button(win, text="Click Me!", command=click_me)
action.grid(column=2, row=1)
```

3. Create a new `ttk.Label` widget:

```
ttk.Label(win, text="Choose a number:").grid(column=1, row=0)
```

4. Create a new `ttk.Combobox` widget:

```
number_chosen = ttk.Combobox(win, width=12, textvariable=number)
```

5. Assign values to the `Combobox` widget:

```
number_chosen['value'] = (1, 2, 4, 42, 100)
```

6. Place the `Combobox` widget into column 1:

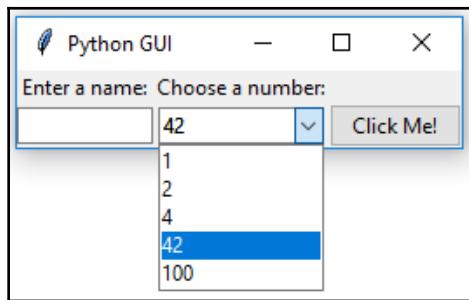
```
number_chosen.grid(column=1, row=1)  
number_chosen.current(0)
```

The preceding steps produce the following code (`GUI_combobox_widget.py`):

```
31 # Adding a Textbox Entry widget  
32 name = tk.StringVar()  
33 name_entered = ttk.Entry(win, width=12, textvariable=name)  
34 name_entered.grid(column=0, row=1) # column 0  
35  
36 # Adding a Button  
37 action = ttk.Button(win, text="Click Me!", command=click_me)  
38 action.grid(column=2, row=1) # <= change column to 2  
39  
40 ttk.Label(win, text="Choose a number:").grid(column=1, row=0)  
41 number = tk.StringVar()  
42 number_chosen = ttk.Combobox(win, width=12, textvariable=number)  
43 number_chosen['values'] = (1, 2, 4, 42, 100)  
44 number_chosen.grid(column=1, row=1) # <= Combobox in column 1  
45 number_chosen.current(0)  
46  
47 name_entered.focus() # Place cursor into name Entry  
48 #=====  
49 # Start GUI  
50 #=====  
51 win.mainloop()
```

7. Run the code.

The code, when added to the previous recipes, creates the following GUI. Note how, in line 43 in the preceding code, we assigned a tuple with default values to the combobox. These values then appear in the drop-down box. We can also change them if we like (by typing in different values when the application is running):



Let's go behind the scenes to understand the code better.

How it works...

Line 40 adds a second label to match the newly created combobox (created in line 42). Line 41 assigns the value of the box to a variable of a special `tkinter` type `StringVar`, as we did in a previous recipe.

Line 44 aligns the two new controls (label and combobox) within our previous GUI layout, and line 45 assigns a default value to be displayed when the GUI first becomes visible. This is the first value of the `number_chosen['values']` tuple, the string "1". We did not place quotes around our tuple of integers in line 43, but they were cast into strings because, in line 41, we declared the values to be of the `tk.StringVar` type.

The preceding screenshot shows the selection made by the user is 42. This value gets assigned to the `number` variable.

If 100 is selected in the combobox, the value of the `number` variable becomes 100.

Line 42 binds the value selected in the combobox to the `number` variable via the `textvariable` attribute.

There's more...

If we want to restrict the user to only being able to select the values we have programmed into the Combobox widget, we can do it by passing the `state` attribute into the constructor. Modify *line 42* as follows:

1. Start with the `GUI_combobox_widget.py` module and save it as `GUI_combobox_widget_READONLY.py`.
2. Set the `state` attribute when creating the Combobox widget:

```
number_chosen = ttk.Combobox(win, width=12, textvariable=number,  
state='readonly')
```

The preceding steps produce the following code
(`GUI_combobox_widget_READONLY.py`):

```
40  ttk.Label(win, text="Choose a number:").grid(column=1, row=0)  
41  number = tk.StringVar()  
42  number_chosen = ttk.Combobox(win, width=12, textvariable=number, state='readonly')  
43  number_chosen['values'] = (1, 2, 4, 42, 100)  
44  number_chosen.grid(column=1, row=1)  
45  number_chosen.current(0)
```

3. Run the code.

Now, users can no longer type values into the Combobox widget.

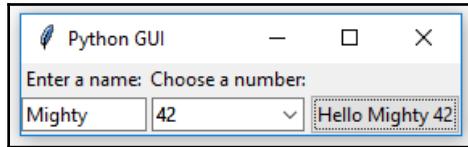
We can display the value chosen by the user by adding the following line of code to our button click event callback function:

1. Start with the `GUI_combobox_widget_READONLY.py` module and save it as `GUI_combobox_widget_READONLY_plus_display_number.py`.
2. Extend the button click event handler by using the `get()` method on the `name` variable, use concatenation (`+ ' ' +`), and also get the number from the `number_chosen` variable (also calling the `get()` method on it):

```
def click_me():  
    action.configure(text='Hello ' + name.get() + ' ' +  
    number_chosen.get())
```

3. Run the code.

After choosing a number, entering a name, and then clicking the button, we get the following GUI result, which now also displays the number selected next to the name entered (`GUI_combobox_widget_READONLY_plus_display_number.py`):



We've successfully learned how to add combobox widgets. Now, let's move on to the next recipe.

Creating a check button with different initial states

In this recipe, we will add three check button widgets, each with a different initial state:

- The first is disabled and has a checkmark in it. The user cannot remove this checkmark as the widget is disabled.
- The second check button is enabled, and by default has no checkmark in it, but the user can click it to add a checkmark.
- The third check button is both enabled and checked by default. The users can uncheck and recheck the widget as often as they like.

Getting ready

This recipe extends the previous recipe, *Creating combobox widgets*.

How to do it...

Here is the code for creating three check button widgets that differ in their states:

1. Start with the `GUI_combobox_widget_READONLY_plus_display_number.py` module and save it as `GUI_checkbutton_widget.py`.
2. Create three `tk.IntVar` instances and save them in local variables:

```
chVarDis = tk.IntVar()  
chVarUn = tk.IntVar()  
chVarEn = tk.IntVar()
```

3. Set the `text` attributes for each of the Combobox widgets we are creating:

```
text="Disabled"  
text="UnChecked"  
text="Enabled"
```

4. Set their state to deselect/select:

```
check1.select()  
check2.deselect()  
check3.select()
```

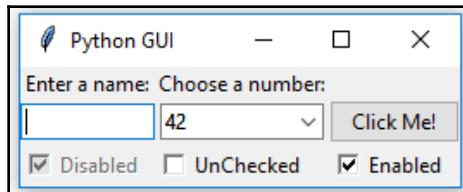
5. Use `grid` to lay them out:

```
check1.grid(column=0, row=4, sticky=tk.W)  
check2.grid(column=1, row=4, sticky=tk.W)  
check3.grid(column=2, row=4, sticky=tk.W)
```

The preceding steps will finally produce the following code (GUI_checkbutton_widget.py):

```
35 # Adding a Button
36 action = ttk.Button(win, text="Click Me!", command=click_me)
37 action.grid(column=2, row=1)
38
39 # Creating a label and a Combobox
40 ttk.Label(win, text="Choose a number:").grid(column=1, row=0)
41 number = tk.StringVar()
42 number_chosen = ttk.Combobox(win, width=12, textvariable=number, state='readonly')
43 number_chosen['values'] = (1, 2, 4, 42, 100)
44 number_chosen.grid(column=1, row=1)
45 number_chosen.current(0)
46 # Creating three checkbuttons
47 chVarDis = tk.IntVar()
48 check1 = tk.Checkbutton(win, text="Disabled", variable=chVarDis, state='disabled')
49 check1.select()
50 check1.grid(column=0, row=4, sticky=tk.W)
51
52 chVarUn = tk.IntVar()
53 check2 = tk.Checkbutton(win, text="UnChecked", variable=chVarUn)
54 check2.deselect()
55 check2.grid(column=1, row=4, sticky=tk.W)
56
57 chVarEn = tk.IntVar()
58 check3 = tk.Checkbutton(win, text="Enabled", variable=chVarEn)
59 check3.select()
60 check3.grid(column=2, row=4, sticky=tk.W)
61
62 name_entered.focus()      # Place cursor into name Entry
63 ##### Start GUI #####
64 win.mainloop()
```

- Run the module. Running the new code results in the following GUI:



Let's go behind the scenes to understand the code better.

How it works...

Steps 1 to 4 show the details and the screenshot in step 5 displays the important aspects of the code.

In lines 47, 52, and 57, we create three variables of the `IntVar` type. In the line following each of these variables, we create a `Checkbutton` widget, passing in these variables. They will hold the state of the `Checkbutton` widget (unchecked or checked). By default, that is either 0 (unchecked) or 1 (checked), so the type of the variable is a `tkinter` integer.

We place these `Checkbutton` widgets in our main window, so the first argument passed into the constructor is the parent of the widget, in our case, `win`. We give each `Checkbutton` widget a different label via its `text` attribute.

Setting the `sticky` property of the grid to `tk.W` means that the widget will be aligned to the west of the grid. This is very similar to Java syntax, and it means that it will be aligned to the left. When we resize our GUI, the widget will remain on the left side and not be moved toward the center of the GUI.

Lines 49 and 59 place a checkmark into the `Checkbutton` widget by calling the `select()` method on these two `Checkbutton` class instances.

We continue to arrange our widgets using the grid layout manager, which will be explained in more detail in Chapter 2, *Layout Management*.

We've successfully learned how to create a check button with different initial states. Now, let's move on to the next recipe.

Using radio button widgets

In this recipe, we will create three radio button widgets. We will also add some code that changes the color of the main form, depending upon which radio button is selected.

Getting ready

This recipe extends the previous recipe, *Creating a check button with different initial states*.

How to do it...

We add the following code to the previous recipe:

1. Start with the `GUI_checkbutton_widget.py` module and save it as `GUI_radiobutton_widget.py`.
2. Create three module-level global variables for the color names:

```
COLOR1 = "Blue"  
COLOR2 = "Gold"  
COLOR3 = "Red"
```

3. Create a callback function for the radio buttons:

```
if radSel == 1: win.configure(background=COLOR1)  
elif radSel == 2: win.configure(background=COLOR2)  
elif radSel == 3: win.configure(background=COLOR3)
```

4. Create three `tk` radio buttons:

```
rad1 = tk.Radiobutton(win, text=COLOR1, variable=radVar, value=1,  
                      command=radCall)  
rad2 = tk.Radiobutton(win, text=COLOR2, variable=radVar, value=2,  
                      command=radCall)  
rad3 = tk.Radiobutton(win, text=COLOR3, variable=radVar, value=3,  
                      command=radCall)
```

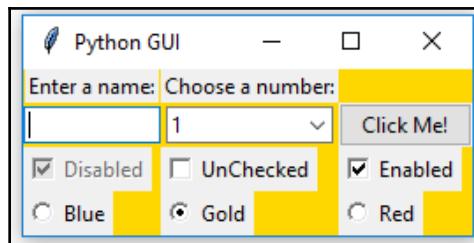
5. Use the grid layout to position them:

```
rad1.grid(column=0, row=5, sticky=tk.W, columnspan=3)  
rad2.grid(column=1, row=5, sticky=tk.W, columnspan=3)  
rad3.grid(column=2, row=5, sticky=tk.W, columnspan=3)
```

The preceding steps will finally produce the following code (GUI_radiobutton_widget.py):

```
74 # Radiobutton Globals
75 COLOR1 = "Blue"
76 COLOR2 = "Gold"
77 COLOR3 = "Red"
78
79 # Radiobutton Callback
80 def radCall():
81     radSel=radVar.get()
82     if radSel == 1: win.configure(background=COLOR1)
83     elif radSel == 2: win.configure(background=COLOR2)
84     elif radSel == 3: win.configure(background=COLOR3)
85
86 # create three Radiobuttons using one variable
87 radVar = tk.IntVar()
88
89 rad1 = tk.Radiobutton(win, text=COLOR1, variable=radVar, value=1, command=radCall)
90 rad1.grid(column=0, row=5, sticky=tk.W, columnspan=3)
91
92 rad2 = tk.Radiobutton(win, text=COLOR2, variable=radVar, value=2, command=radCall)
93 rad2.grid(column=1, row=5, sticky=tk.W, columnspan=3)
94
95 rad3 = tk.Radiobutton(win, text=COLOR3, variable=radVar, value=3, command=radCall)
96 rad3.grid(column=2, row=5, sticky=tk.W, columnspan=3)
97
98 name_entered.focus()      # Place cursor into name Entry
99 #####
100 # Start GUI
101 #####
102 win.mainloop()
```

6. Run the code. Running this code and selecting the radio button named **Gold** creates the following window:



Let's go behind the scenes to understand the code better.

How it works...

In lines 75-77, we create some module-level global variables that we will use in the creation of each radio button, as well as in the callback function that creates the action of changing the background color of the main form (using the `win` instance variable).

We are using global variables to make it easier to change the code. By assigning the name of the color to a variable and using this variable in several places, we can easily experiment with different colors. Instead of doing a global search and replace of the hardcoded string (which is prone to errors), we just need to change one line of code and everything else will work. This is known as the **DRY principle**, which stands for **Don't Repeat Yourself**. This is an OOP concept that we will use in the later recipes of the book.



The names of the colors we are assigning to the variables (`COLOR1`, `COLOR2`, ...) are `tkinter` keywords (technically, they are *symbolic names*). If we use names that are not `tkinter` color keywords, then the code will not work.

Line 80 is the *callback function* that changes the background of our main form (`win`) depending upon the user's selection.

In line 87, we create a `tk.IntVar` variable. What is important about this is that we create only one variable to be used by all three radio buttons. As can be seen from the screenshot, no matter which radio buttons we select, all the others will automatically be unselected for us.

Lines 89 to 96 create the three radio buttons, assigning them to the main form, passing in the variable to be used in the callback function that creates the action of changing the background of our main window.



While this is the first recipe that changes the color of a widget, quite honestly, it looks a bit ugly. A large portion of the following recipes in this book explain how to make our GUI look truly amazing.

There's more...

Here is a small sample of the available symbolic color names that you can look up in the official TCL documentation at <http://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm>:

Name	Red	Green	Blue
alice blue	240	248	255
AliceBlue	240	248	255
Blue	0	0	255
Gold	255	215	0
Red	255	0	0

Some of the names create the same color, so `alice blue` creates the same color as `AliceBlue`. In this recipe, we used the symbolic names `Blue`, `Gold`, and `Red`.

We've successfully learned how to use radio button widgets. Now, let's move on to the next recipe.

Using scrolled text widgets

`ScrolledText` widgets are much larger than simple `Entry` widgets and span multiple lines. They are widgets like Notepad and wrap lines, automatically enabling vertical scroll bars when the text gets larger than the height of the `ScrolledText` widget.

Getting ready

This recipe extends the previous recipe, *Using radio button widgets*. You can download the code for each chapter of this book from <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition/>.

How to do it...

By adding the following lines of code, we create a `ScrolledText` widget:

1. Start with the `GUI_radiobutton_widget.py` module and save it as `GUI_scrolledtext_widget.py`.

2. Import `scrolledtext`:

```
from tkinter import scrolledtext
```

3. Define variables for the width and height:

```
scrol_w = 30  
scrol_h = 3
```

4. Create a `ScrolledText` widget:

```
scr = scrolledtext.ScrolledText(win, width=scrol_w, height=scrol_h,  
wrap=tk.WORD)
```

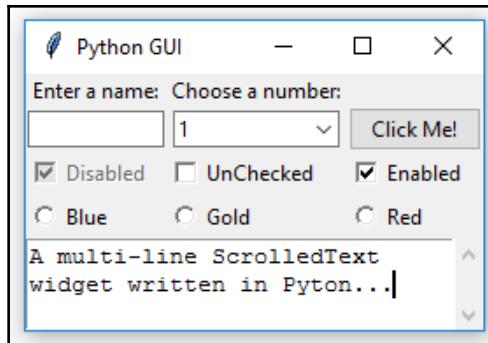
5. Position the widget:

```
scr.grid(column=0, columnspan=3)
```

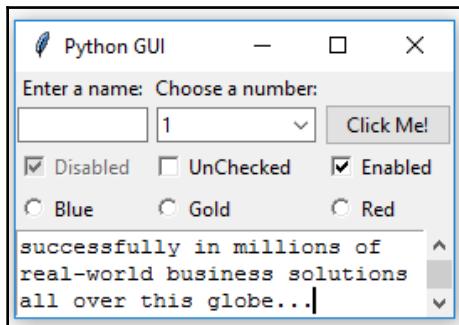
The preceding steps will finally produce the following code
(`GUI_scrolledtext_widget.py`):

```
6#=====  
7# imports  
8#=====  
9import tkinter as tk  
10from tkinter import ttk  
11from tkinter import scrolledtext  
  
99# Using a scrolled Text control  
100scrol_w = 30  
101scrol_h = 3  
102scr = scrolledtext.ScrolledText(win, width=scrol_w, height=scrol_h, wrap=tk.WORD)  
103scr.grid(column=0, columnspan=3)  
104  
105name_entered.focus()      # Place cursor into name Entry  
106#=====  
107# Start GUI  
108#=====  
109win.mainloop()
```

6. Run the code. We can actually type into our widget, and if we type enough words, the lines will automatically wraparound:



Once we type in more words than the height of the widget can display, the vertical scroll bar becomes enabled. This all works out of the box without us needing to write any more code to achieve this:



Let's go behind the scenes to understand the code better.

How it works...

In line 11, we import the module that contains the `ScrolledText` widget class. Add this to the top of the module, just below the other two `import` statements.

Lines 100 and 101 define the width and height of the `ScrolledText` widget we are about to create. These are hardcoded values we are passing into the `ScrolledText` widget constructor in line 102.

These values are *magic numbers* found by experimentation to work well. You might experiment by changing `scol_w` from 30 to 50 and observe the effect!

In line 102, we are also setting a property on the widget by passing in `wrap=tk.WORD`. By setting the `wrap` property to `tk.WORD`, we are telling the `ScrolledText` widget to break lines by words so that we do not wraparound within a word. The default option is `tk.CHAR`, which wraps any character regardless of whether we are in the middle of a word.

The second screenshot shows that the vertical scroll bar moved down because we are reading a longer text that does not entirely fit into the *x, y* dimensions of the `ScrolledText` control we created.

Setting the `columnspan` attribute of the grid layout to 3 for the `ScrolledText` widget makes this widget span all of the three columns. If we do not set this attribute, our `ScrolledText` widget would only reside in column one, which is not what we want.

We've successfully learned how to use scrolled text widgets. Now, let's move on to the next recipe.

Adding several widgets in a loop

So far, we have created several widgets of the same type (for example, a radio button) by basically copying and pasting the same code and then modifying the variations (for example, the column number). In this recipe, we start refactoring our code to make it less redundant.

Getting ready

We are refactoring some parts of the previous recipe's code, *Using scrolled text widgets*, so you need that code for this recipe.

How to do it...

Here's how we refactor our code:

1. Start with the `GUI_scrolledtext_widget.py` module and save it as `GUI_adding_widgets_in_loop.py`.
2. Delete the global name variables and create a Python list instead:

```
colors = ["Blue", "Gold", "Red"]
```

3. Use the `get()` function on the radio button variable:

```
radSel=radVar.get()
```

4. Create logic with an `if ... elif` structure:

```
if radSel == 0: win.configure(background=colors[0])
elif radSel == 1: win.configure(background=color[1])
elif radSel == 2: win.configure(background=color[2])
```

5. Use a loop to create and position the radio buttons:

```
for col in range(3):
    curRad = tk.Radiobutton(win, text=colors[col], variable=radVar,
                           value, command=radCall)
    curRad.grid(column=col, row=5, sticky=tk.W)
```

6. Run the code (`GUI_adding_widgets_in_loop.py`):

```
76 # First, we change our Radiobutton global variables into a list
77 colors = ["Blue", "Gold", "Red"]
78
79# We have also changed the callback function to be zero-based, using the list
80# instead of module-level global variables
81 # Radiobutton Callback
82def radCall():
83     radSel=radVar.get()
84     if radSel == 0: win.configure(background=colors[0]) # now zero-based
85     elif radSel == 1: win.configure(background=colors[1]) # and using list
86     elif radSel == 2: win.configure(background=colors[2])
87
88 # create three Radiobuttons using one variable
89 radVar = tk.IntVar()
90
91 # Next we are selecting a non-existing index value for radVar
92 radVar.set(99)
93
94 # Now we are creating all three Radiobutton widgets within one loop
95 for col in range(3):
96     curRad = tk.Radiobutton(win, text=colors[col], variable=radVar,
97                            value=col, command=radCall)
98     curRad.grid(column=col, row=5, sticky=tk.W)
99
```

Running this code will create the same window as before, but our code is much cleaner and easier to maintain. This will help us when we expand our GUI in the coming recipes.

How it works...

In *line 77*, we have turned our global variables into a list. In *line 89*, we set a default value to the `tk.IntVar` variable that we named `radVar`. This is important because, while in the previous recipe we had set the value for radio button widgets to start at 1, in our new loop it is much more convenient to use Python's zero-based indexing. If we did not set the default value to a value outside the range of our radio button widgets, one of the radio buttons would be selected when the GUI appears. While this in itself might not be so bad, *it would not trigger the callback* and we would end up with a radio button selected that does not do its job (that is, change the color of the main win form).

In *line 95*, we replace the three previously hardcoded creations of the radio button widgets with a loop that does the same. It is just more concise (fewer lines of code) and much more maintainable. For example, if we want to create 100 instead of just 3 radio button widgets, all we have to change is the number inside Python's range operator. We would not have to type or copy and paste 97 sections of duplicate code, just 1 number.

Line 82 shows the modified callback function.

This recipe concludes the first chapter of this book. All the following recipes in all of the next chapters will build upon the GUIs we have constructed so far, greatly enhancing it.

2

Layout Management

In this chapter, we will explore how to arrange widgets within widgets to create a Python GUI. Learning about the fundamentals of GUI layout design will allow us to create great-looking GUIs. There are certain techniques that will help us achieve this layout design.

The grid layout manager is one of the most important layout tools that we will be using, and is built into `tkinter`. We can very easily create menu bars, tabbed controls (that is, Notebooks), and many more widgets using `tkinter`.

By completing this chapter, you will learn how to arrange your widgets to make your GUI look truly great! Learning layout management is fundamental to GUI design, even if you use other programming languages – but Python truly rocks!

The following screenshot provides an overview of the Python modules that will be used in this chapter:



In this chapter, we will lay out our GUI using Python 3.7 and above. We will provide the following recipes:

- Arranging several labels within a label frame widget
- Using padding to add space around widgets
- How widgets dynamically expand the GUI
- Aligning GUI widgets by embedding frames within frames
- Creating menu bars
- Creating tabbed widgets
- Using the grid layout manager

Arranging several labels within a label frame widget

The `LabelFrame` widget allows us to design our GUI in an organized fashion. We are still using the grid layout manager as our main layout design tool, but by using `LabelFrame` widgets, we get much more control over our GUI's design.

Getting ready

We will start by adding more widgets to our GUI. We will make the GUI fully functional in upcoming recipes. Here, we will start to use the `LabelFrame` widget. We will reuse the GUI from the *Adding several widgets to a loop* recipe in Chapter 1, *Creating the GUI Form and Adding Widgets*.

How to do it...

1. Open `GUI_adding_widgets_in_loop.py` from Chapter 1, *Creating the GUI Form and Adding Widgets*, and save the module as `GUI_LabelFrame_column_one.py`.
2. Create a `ttk.LabelFrame` and position it in the grid:

```
buttons_frame = ttk.LabelFrame(win, text=' Labels in a Frame ')
buttons_frame.grid(column=0, row=7)
# button_frame.grid(column=1, row=7)
```

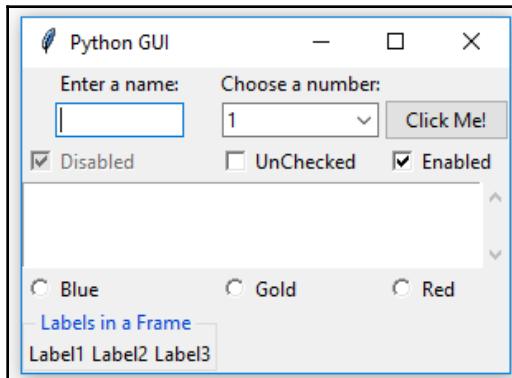
3. Create three `ttk.Label`s, set their `text` attributes, and position them in the grid:

```
ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0,
sticky=tk.W)
ttk.Label(buttons_frame, text="Label2").grid(column=1, row=0,
sticky=tk.W)
ttk.Label(buttons_frame, text="Label3").grid(column=2, row=0,
sticky=tk.W)
```

The preceding instructions produce the following code from the `GUI_LabelFrame_column_one.py` file:

```
108 # Create a container to hold labels
109 buttons_frame = ttk.LabelFrame(win, text=' Labels in a Frame ')
110 buttons_frame.grid(column=0, row=7)
111 # buttons_frame.grid(column=1, row=7) # now in col 1
112
113 # Place labels into the container element
114 ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0, sticky=tk.W)
115 ttk.Label(buttons_frame, text="Label2").grid(column=1, row=0, sticky=tk.W)
116 ttk.Label(buttons_frame, text="Label3").grid(column=2, row=0, sticky=tk.W)
117
118 name_entered.focus() # Place cursor into name Entry
119 #=====
120 # Start GUI
121 #=====
122 win.mainloop()
```

- Run the code. It will result in the following GUI:



Uncomment line 111 and notice the different alignment of `LabelFrame`.



In addition, we can easily align the labels vertically by changing our code. To do this perform the following steps:

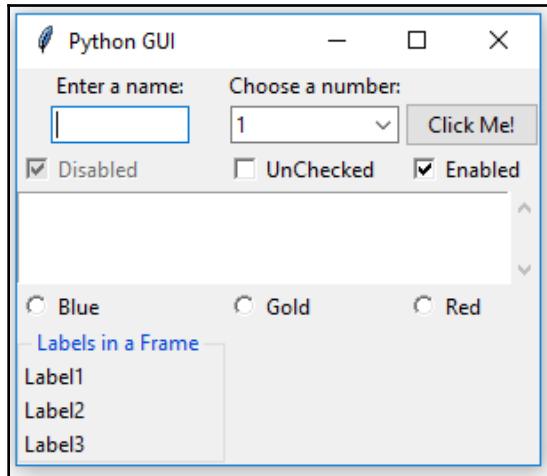
1. Open `GUI_LabelFrame_column_one.py` and save the module as `GUI_LabelFrame_column_one_vertical.py`.
2. Change the column and row values, as follows:

```
ttk.Label(button_frame, text="Label1").grid(column=0, row=0)
ttk.Label(button_frame, text="Label2").grid(column=0, row=1)
ttk.Label(button_frame, text="Label3").grid(column=0, row=2)
```

The only change we had to make was in the column and row numbering.



3. Run the `GUI_LabelFrame_column_one_vertical.py` file. Now the GUI label frame will look as follows:



Now let's go behind the scenes to understand the code better.

How it works...

In line 109, we create our first `ttk LabelFrame` widget and assign the resulting instance to the `buttons_frame` variable. The parent container is `win`, which is our main window.

In lines 114 to 116, we create labels and place them into a `LabelFrame`. `buttons_frame` is the parent of the labels. We use the important grid layout tool to arrange the labels within `LabelFrame`. The column and row properties of this layout manager give us the power to control our GUI layout.



The parent of our labels is the `buttons_frame` instance variable of `LabelFrame`, not the `win` instance variable of the main window. We can see the beginning of a layout hierarchy here.

We can see how easy it is to change our layout via the column and row properties. Note how we change the column to 0, and how we layer our labels vertically by numbering the row values sequentially.



The name `ttk` stands for **themed tk**. The **tk-themed** widget set was introduced in Tk 8.5.

We've successfully learned how to arrange several labels within a `LabelFrame` widget.

See also...

In the *Aligning GUI widgets by embedding frames within frames* recipe, we will embed `LabelFrame` widgets within `LabelFrame` widgets, nesting them to control our GUI layout.

Now let's move on to the next recipe.

Using padding to add space around widgets

Our GUI is coming along nicely. Next, we will improve the visual aspects of our widgets by adding a little space around them so that they can breathe.

Getting ready

While `tkinter` might have had a reputation for creating not-so-pretty GUIs, this has dramatically changed since version 8.5.

To better understand the major improvements to Tk, the following is a quote from the official website; you can find it at the following link:

<https://tkdocs.com/tutorial/onepage.html>:



"This tutorial is designed to help people get up to speed quickly with building mainstream desktop graphical user interfaces with Tk, and in particular Tk 8.5 and 8.6. Tk 8.5 was an incredibly significant milestone release and a significant departure from the older versions of Tk which most people know and recognize."

You just have to know how to use the tools and techniques that are available. That's what we will do next.



`tkinter` version 8.6 ships with Python 3.7. There's no need to install anything other than Python in order to use it.

A simple way of adding spacing around widgets will be shown first, and then we will use a loop to achieve the same thing in a much better way.

Our `LabelFrame` looks a bit tight as it blends into the main window toward the bottom. Let's fix this now.

How to do it...

Follow these steps to add padding around the widgets:

1. Open `GUI_LabelFrame_column_one.py` and save it as `GUI_add_padding.py`.
2. Add `padx` and `pady` to the `grid` method:

```
buttons_frame.grid(column=0, row=7, padx=20, pady=40)  
# padx, pady
```

3. Run the code. Now our `LabelFrame` has some breathing space. We can see this in the following screenshot:



We can use a loop to add space around the labels contained within `LabelFrame`. Follow these steps to do so:

1. Open `GUI_add_padding.py` and save it as `GUI_add_padding_loop.py`.
2. Add the following loop below the creation of the three Labels:

```
for child in buttons_frame.winfo_children():
    child.grid_configure(padx=8, pady=4)
```

The preceding instructions produce the following code:

```
113 # Place labels into the container element
114 ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0)
115 ttk.Label(buttons_frame, text="Label2").grid(column=0, row=1)
116 ttk.Label(buttons_frame, text="Label3").grid(column=0, row=2)
117
118 for child in buttons_frame.winfo_children():
119     child.grid_configure(padx=8, pady=4)
120
121 name_entered.focus()      # Place cursor into name Entry
122 #=====
123 # Start GUI
124 #=====
125 win.mainloop()
```

3. Run the `GUI_add_padding_loop.py` file code. Now the labels within the `LabelFrame` widget have some space around them too:

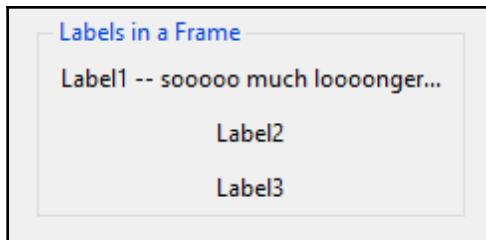


To see this effect better, let's do the following:

1. Open `GUI_add_padding_loop.py` and save it as `GUI_long_label.py`.
2. Change the text of `Label1`, like so:

```
ttk.Label(buttons_frame, text="Label1 -- sooooo much  
loooonger...").grid(column=0, row=0)
```

3. Run the code. This will generate what's shown in the following screenshot, which shows our GUI. Note how there is now space to the right of the long label, next to the dots. The last dot doesn't touch `LabelFrame`, which it otherwise would have without the added space:



We can also remove the `LabelFrame` name to see the effect `padx` has on the position of our labels. Let's get started:

1. Open `GUI_add_padding_loop.py` and save it as `GUI_LabelFrame_no_name.py`.
2. When creating the button, set the text attribute to an empty string:

```
buttons_frame = ttk.LabelFrame(win, text='') # no  
LabelFrame name
```

3. Run the code. By setting the `text` attribute to an empty string, we remove the name that was previously displayed for `LabelFrame`. This can be seen in the following screenshot:



Now let's go behind the scenes to understand the code better.

How it works...

In `tkinter`, adding space horizontally and vertically is done by using the built-in `padx` and `pady` attributes. These can be used to add space around many widgets, improving horizontal and vertical alignments, respectively. We hard-coded 20 pixels of space to the left and right of `LabelFrame`, and we added 40 pixels to the top and bottom of the frame. Now our `LabelFrame` stands out better than it did before.

The `grid_configure()` function allows us to modify the UI elements before the main loop displays them. So, instead of hard-coding values when we first create a widget, we can work on our layout and then arrange spacing toward the end of our file, just before the GUI is created. This is a neat technique to know about.

The `winfo_children()` function returns a list of all the children belonging to the `buttons_frame` variable. This allows us to loop through them and assign the padding to each label.



One thing to notice is that the spacing to the right of the labels isn't really visible. This is because the title of `LabelFrame` is longer than the names of the labels. We suggest you experiment with this by making the label names longer.

We've successfully learned how to add space around widgets using padding. Now let's move on to the next recipe.

Dynamically expanding the GUI using widgets

You may have noticed from the previous screenshots and by running the preceding code that the widgets can extend themselves to take up as much space as they need in order to visually display their text.

Java introduced the concept of dynamic GUI layout management. In comparison, visual development IDEs, such as VS.NET, lay out the GUI in a visual manner, and basically hard-code the x and y coordinates of the UI elements.



Using `tkinter`, this dynamic capability creates both an advantage and a little bit of a challenge because, sometimes, our GUI dynamically expands when we would like it not to be so dynamic! Well, we are dynamic Python programmers, so we can figure out how to make the best use of this fantastic behavior!

Getting ready

At the beginning of the previous recipe, *Using padding to add space around widgets*, we added a `LabelFrame` widget. This moved some of our controls to the center of column 0. We might not want this modification in our GUI layout. We will explore some ways to solve this in this recipe.

First, let's take a look at the subtle details that are going on in our GUI layout in order to understand it better.

We are using the `grid` layout manager widget, which places our widgets in a zero-based grid. This is very similar to an Excel spreadsheet or a database table.

The following is an example of a grid layout manager with two rows and three columns:

Row 0; Col 0	Row 0; Col 1	Row 0; Col 2
Row 1; Col 0	Row 1; Col 1	Row 1; Col 2



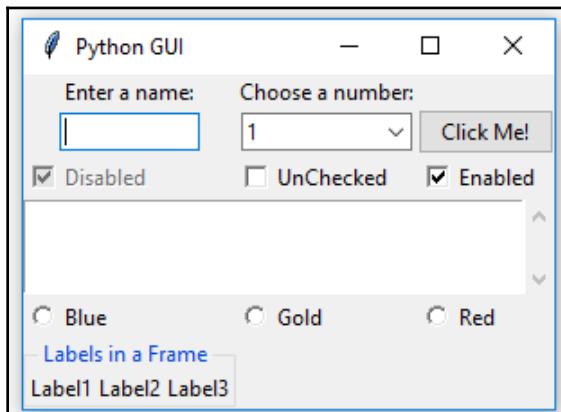
Using the grid layout manager, the width of any given column is determined by the longest name or widget in that column. This, in turn, affects all of the rows.

By adding our `LabelFrame` widget and giving it a title that is longer than a hard-coded size widget, we dynamically move those widgets to the center of column 0. By doing so, we are adding space on the left- and right-hand side of those widgets.

Incidentally, because we used the `sticky` property for the `Checkbutton` and `ScrolledText` widgets, those remain attached to the left-hand side of the frame.

Let's take a look at the screenshot from the first recipe in this chapter, *Arranging several labels within a label frame widget*, in more detail.

Since the text property of `LabelFrame`, which is displayed as the title of `LabelFrame`, is longer than both our `Enter a name:` label and the text box entry below it, those two widgets are dynamically centered within the new width of column 0, as shown in the following screenshot:





Notice how both the label and the entry below it are no longer positioned on the left but have been moved to the center within the grid column.

We added the following code to `GUI_LabelFrame_no_name.py` to create a `LabelFrame` and then placed labels in this frame to stretch both the `Label` frame and the widgets contained therein:

```
buttons_frame = ttk.LabelFrame(win, text='Labels in a Frame')
buttons_frame.grid(column=0, row=7)
```

The `Checkbutton` and `Radiobutton` widgets in column 0 did not get centered because we used the `sticky=tk.W` attribute when we created those widgets.

For the `ScrolledText` widget, we also used `sticky=tk.WE`, which binds the widget to both the west (the left) and east (the right) side of the frame.



The `sticky` attribute is available in `tkinter` and aligns widgets within the `grid` control.

How to do it...

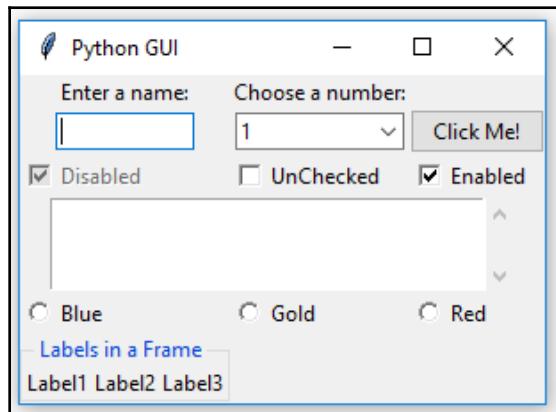
Perform the following steps to complete this recipe:

1. Open `GUI_arranging_labels.py` and save it as `GUI_remove_sticky.py`.
2. Remove the `sticky` attribute from the `ScrolledText` widget and observe the effect this change has.

The preceding instructions produce the following code. Notice how the original `src.grid(...)` is now commented out and the new `src.grid(...)` no longer has the `sticky` attribute:

```
# Using a scrolled Text control
scrol_w = 30
scrol_h = 3
scr = scrolledtext.ScrolledText(win, width=scrol_w, height=scrol_h, wrap=tk.WORD)
#### scr.grid(column=0, row=5, sticky='WE', columnspan=3)
scr.grid(column=0, row=5, columnspan=3) # sticky property removed
```

3. Run the code. Now our GUI has a new space around the `ScrolledText` widget, both on the left- and right-hand sides. Because we used the `columnspan=3` property, our `ScrolledText` widget still spans all three columns. This is shown in the following screenshot:

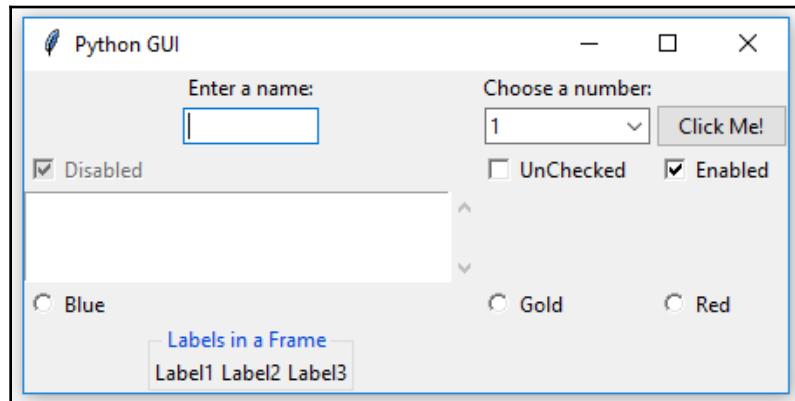


Using `columnspan` is necessary to arrange our GUI in the way we desire it to look.

Let's take a look at how *not* using the `columnspan` attribute could screw up our nice GUI design by doing the following modifications:

1. Open `GUI_remove_sticky.py` and save it as `GUI_remove_columnspan.py`.
2. If we remove `columnspan=3`, we'll get the GUI that's shown in the following screenshot, which is not what we want. Now `ScrolledText` only occupies column 0 and, because of its size, stretches the layout.

- Run the `GUI_remove_columnspan.py` file and observe the output:

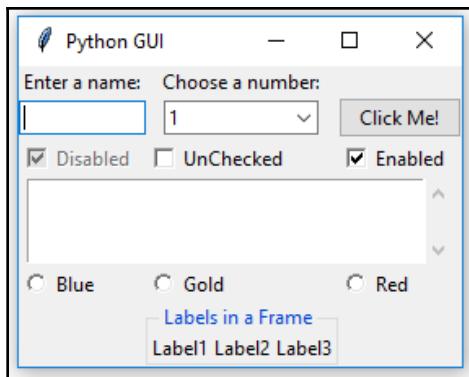


One way to get our layout back to where it was before adding `LabelFrame` is to adjust the grid column position. Let's get started:

- Open `GUI_remove_columnspan.py` and save it as `GUI_LabelFrame_column_one.py`.
- Change the column value from 0 to 1:

```
buttons_frame.grid(column=1, row=7)      # now in col 1
```

- Run the code. Now our GUI will look as follows:



Let's go behind the scenes to understand the code better.

How it works...

Because we are still using individual widgets, our layout can get messed up. By moving the column value of `LabelFrame` from 0 to 1, we were able to get the controls back to where they used to be and where we prefer them to be. The left-most label, text, `Checkbutton`, `ScrolledText`, and `Radiobutton` widgets are now located where we intended them to be. The second label and the `Entry` text located in column 1 aligned themselves to the center of the length of the **Labels in a Frame** widget, so we basically moved our alignment challenge one column to the right. It is not so visible now because the size of the **Choose a number:** label is almost the same as the size of the **Labels in a Frame** title, and so the column's width was already close to the new width that was generated by `LabelFrame`.

There's more...

In the next recipe, *Aligning GUI widgets by embedding frames within frames*, we will embed frames within frames to avoid the accidental misalignment of widgets we just experienced in this recipe.

We've successfully learned how to dynamically expand the GUI using widgets. Now let's move on to the next recipe.

Aligning GUI widgets by embedding frames within frames

We'll have better control of our GUI layout if we embed frames within frames. This is what we will do in this recipe.

Getting ready

The dynamic behavior of Python and its GUI modules can create challenges when we want to make our GUI really look the way we want it to. In this recipe, we will embed frames within frames to get more control of our layout. This will establish a stronger hierarchy among the different UI elements, making the visual appearance easier to achieve.

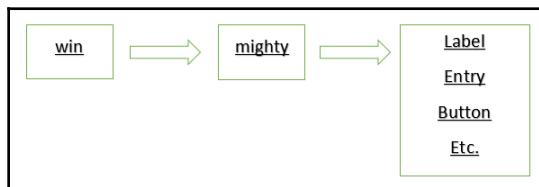
We will continue using the GUI we created in the previous recipe, *Dynamically expanding the GUI using widgets*.

Here, we will create a top-level frame that will contain other frames and widgets. This will help us get our GUI layout just the way we want.

To do so, we will have to embed our current controls within a central frame called `ttk.LabelFrame`. This frame, `ttk.LabelFrame`, is the child of the main parent window, and all the controls will be the children of this `ttk.LabelFrame`.

So far, we have assigned all the widgets to our main GUI frame directly. Now we will only assign `LabelFrame` to our main window. After that, we will make this `LabelFrame` the parent container for all the widgets.

This creates the following hierarchy in our GUI layout:



In the preceding diagram, `win` is the variable that holds a reference to our main `tkinter` window frame, `mighty` is the variable that holds a reference to our `LabelFrame` and is a child of the main window frame (`win`), and `Label` and all the other widgets are now placed into the `LabelFrame` container (`mighty`).

How to do it...

Perform the following steps to complete this recipe:

1. Open `GUI_LabelFrame_column_one.py` and save it as `GUI_embed_frames.py`.
2. Add the following code toward the top of our Python module:

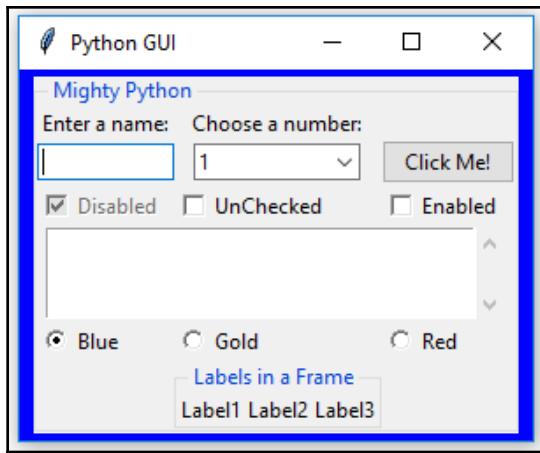
```
mighty = ttk.LabelFrame(win, text=' Mighty Python ')
mighty.grid(column=0, row=0, padx=8, pady=4)
```

Next, we will modify the following controls to use `mighty` as the parent, replacing `win`.

3. Change the Label parent from `win` to `mighty`:

```
a_label = ttk.Label(mighty, text="Enter a name:")
a_label.grid(column=0, row=0)
```

4. Run the `GUI_embed_frames.py` file. This results in the GUI shown in the following screenshot:

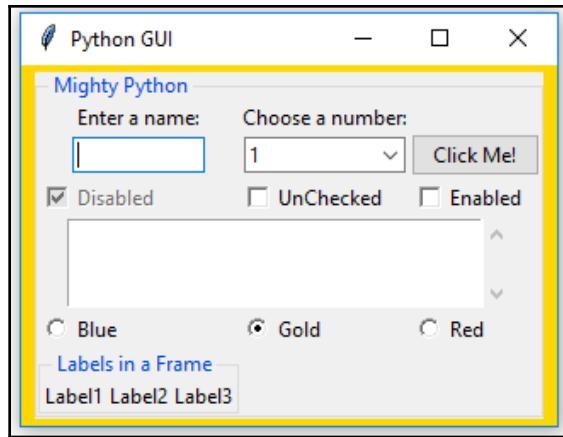


Note how all the widgets are now contained in the **Mighty Python** `LabelFrame`, which surrounds all of them with a barely visible thin line. Next, we can reset the **Labels in a Frame** widget to the left without messing up our GUI layout:

1. Open `GUI_embed_frames.py` and save it as `GUI_embed_frames_align.py`.
2. Change `column` to 0:

```
buttons_frame = ttk.LabelFrame(mighty, text=' Labels in a Frame ')
buttons_frame.grid(column=0, row=7)
```

3. Run the `GUI_embed_frames_align.py` file. This results in the GUI shown in the following screenshot:



Oops – maybe not. While our frame-within-another-frame aligned nicely to the left, it pushed our top widgets to the center (the default setting).

To align them to the left, we have to force our GUI layout by using the `sticky` property. By assigning it '`W`' (west), we can force the widget to be left-aligned. Perform the following steps:

1. Open `GUI_embed_frames_align.py` and save it as `GUI_embed_frames_align_west.py`.
2. Add the `sticky` attribute to the label:

```
a_label = ttk.Label(mighty, text="Enter a name:")
a_label.grid(column=0, row=0, sticky='W')
```

3. Run the code. This gives us the following GUI:

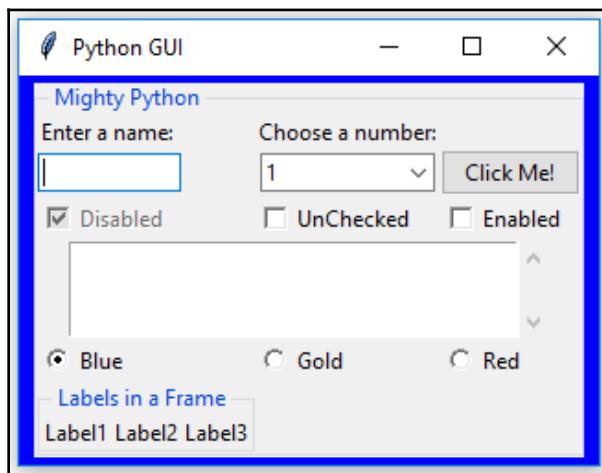


Let's align the `Entry` widget in column 0 to the left:

1. Open `GUI_embed_frames_align_west.py` and save it as `GUI_embed_frames_align_entry_west.py`.
2. Use the `sticky` attribute to align the `Entry` left:

```
name = tk.StringVar()
name_entered = ttk.Entry(mighty, width=12, textvariable=name)
name_entered.grid(column=0, row=1, sticky=tk.W) # align left/West
```

3. Run the `GUI_embed_frames_align_entry_west.py` file. Now both the label and the entry are aligned toward the west (left):



Now let's go behind the scenes to understand the code better.

How it works...

Note how we aligned the label, but not the text box below it. We have to use the `sticky` property for all the controls we want to left-align. We can do that in a loop by using the `winfo_children()` and `grid_configure(sticky='W')` properties, as we did in the *Using padding to add space around widgets* recipe of this chapter.

The `winfo_children()` function returns a list of all the children belonging to the parent. This allows us to loop through all the widgets and change their properties.

Using `tkinter` to force the naming to the left, right, top, or bottom is very similar to Java's West, East, North, and South, which are abbreviated to '`W`', '`E`', and so on. We can also use `tk.W` instead of '`W`'. This requires that we import the `tkinter` module aliased as `tk`.



In a previous recipe, we combined '`W`' and '`E`' to make our `ScrolledText` widget attach itself both to the left- and right-hand sides of its container. The result of combining '`W`' and '`E`' was '`WE`'. We can add more combinations as well: '`NSE`' will stretch our widget to the top, bottom, and right-hand side. If we only have one widget in our form, for example, a button, we can make it fill in the entire frame by using all the options, that is, '`NSWE`'. We can also use tuple syntax: `sticky=(tk.N, tk.S, tk.W, tk.E)`.

To obviate the influence that the length of our **Labels in a Frame** `LabelFrame` has on the rest of our GUI layout, we must not place this `LabelFrame` into the same `LabelFrame` as the other widgets. Instead, we need to assign it directly to the main GUI form (`win`).

We've successfully learned how to align the GUI widget by embedding frames with frames. Now let's move on to the next recipe.

Creating menu bars

In this recipe, we will add a menu bar to our main window, add menus to the menu bar, and then add menu items to the menus.

Getting ready

We will start by learning how to add a menu bar, several menus, and a few menu items. In the beginning, clicking on a menu item will have no effect. We will add functionality to the menu items later, for example, closing the main window when clicking the **Exit** menu item and displaying a **Help | About** dialog.

We will continue to extend the GUI we created in the previous recipe, *Aligning GUI widgets by embedding frames within frames*.

How to do it...

To create a menu bar, follow these steps:

1. Open `GUI_embed_frames_align_entry_west.py` and save it as `GUI_menu_bar_file.py`.
2. Import the `Menu` class from `tkinter`:

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
```

3. Next, we will create the menu bar. Add the following code toward the bottom of the module, just above where we created the main event loop:

```
# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Create menu and add menu items
file_menu = Menu(menu_bar) # create File menu
file_menu.add_command(label="New") # add File menu item
```

The preceding instructions produce the following code from the `GUI_menu_bar_file.py` file:

```
118 # Creating a Menu Bar
119 menu_bar = Menu(win)
120 win.config(menu=menu_bar)
121
122 # Create menu and add menu items
123 file_menu = Menu(menu_bar) # create File menu
124 file_menu.add_command(label="New") # add File menu item
```

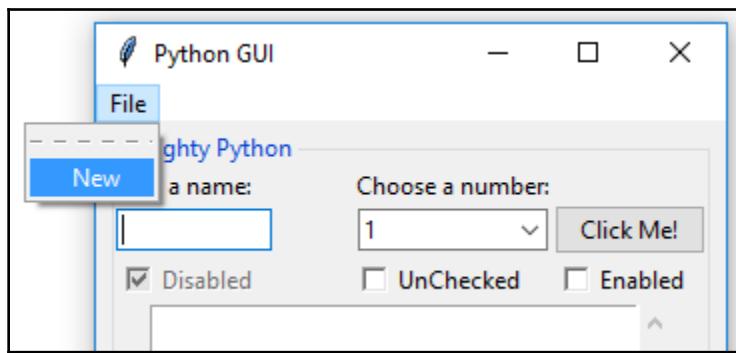
In line 119, we are calling the constructor of the imported `Menu` module class and passing in our main GUI instance, `win`. We save an instance of the `Menu` object in the `menu_bar` variable. In line 120, we configure our GUI to use our newly created `Menu` as the menu for our GUI.

To make this work, we also have to add the menu to the menu bar and give it a label.

4. The menu item was already added to the menu, but we still have to add the menu to the menu bar:

```
menu_bar.add_cascade(label="File", menu=file_menu) # add File menu  
to menu bar and give it a label
```

5. Running the preceding code adds a menu bar with a menu that has a menu item. This is shown in the following screenshot:



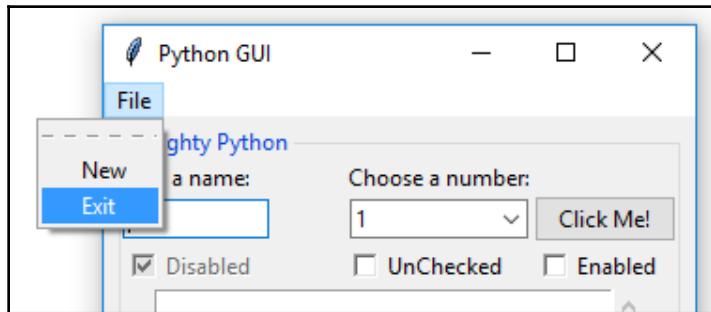
If this `tkinter` menu bar syntax seems a little bit confusing, don't worry. This is just the syntax of `tkinter` for creating a menu bar. It isn't very Pythonic.

Next, we'll add a second menu item to the first menu that we added to the menu bar. This can be done by performing the following steps:

1. Open `GUI_menubar_file.py` and save it as `GUI_menubar_exit.py`.
2. Add the **Exit** menu item:

```
file_menu.add_command(label="Exit")
```

3. Running the preceding code produces the following result, that is, `GUI_menuBar_exit.py`:

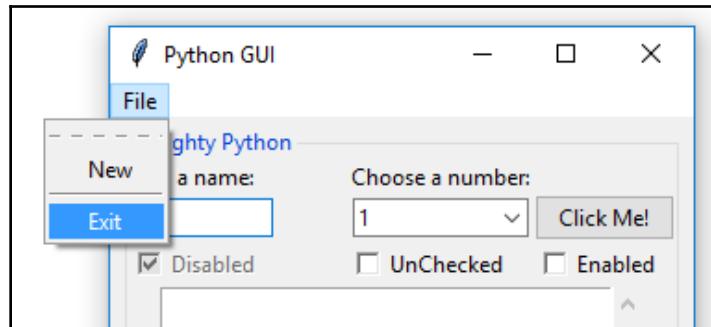


We can add separator lines between the menu items by adding a line of code in-between the existing menu items. This can be done by performing the following steps:

1. Open `GUI_menuBar_exit.py` and save it as `GUI_menuBar_separator.py`.
2. Add a separator, as follows:

```
file_menu.add_separator()
```

3. Run the preceding code. In the following screenshot, we can see that a separator line has been added in-between our two menu items:

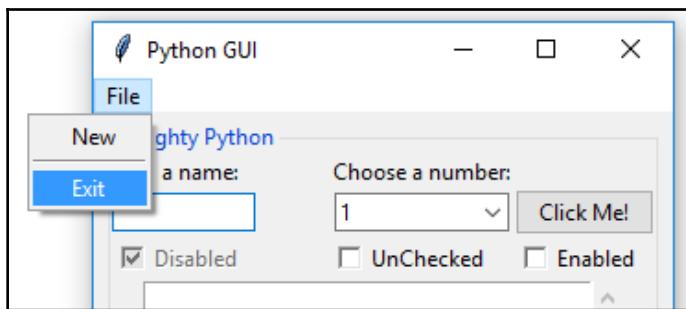


By passing in the `tearoff` property to the constructor of the menu, we can remove the first dashed line that, by default, appears above the first menu item in a menu. This can be done by performing the following steps:

1. Open `GUI_menubar_separator.py` and save it as `GUI_menubar_tearoff.py`.
2. Set the `tearoff` attribute to 0:

```
file_menu = Menu(menu_bar, tearoff=0)
```

3. Run the preceding code. In the following screenshot, the dashed line no longer appears, and our GUI looks so much better:



Next, we'll add a second menu, `Help`, which will be placed horizontally, to the right of the first menu. We'll give it one menu item, named `About`, and add this second menu to the menu bar.

File and Help | About are very common Windows GUI layouts we are all familiar with, and we can create these same menus using Python and `tkinter`:

1. Open `GUI_menubar_tearoff.py` and save it as `GUI_menubar_help.py`.
2. Add a second menu with a menu item:

```
help_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Help", menu=help_menu)
help_menu.add_command(label="About")
```

The preceding instructions produce the following code, which can be found in the `GUI_menuBar_help.py` file:

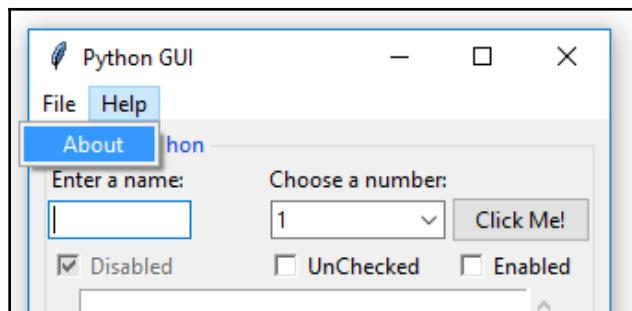
```
# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Add menu items
file_menu = Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New")
file_menu.add_separator()
file_menu.add_command(label="Exit")
menu_bar.add_cascade(label="File", menu=file_menu)

# Add another Menu to the Menu Bar and an item
help_menu = Menu(menu_bar, tearoff=0)
menu_bar.add_cascade(label="Help", menu=help_menu)
help_menu.add_command(label="About")

name_entered.focus()      # Place cursor into name Entry
#=====
# Start GUI
#=====
win.mainloop()
```

3. Run the preceding code. As shown in the following screenshot, we have a second menu with a menu item in the menu bar:



At this point, our GUI has a menu bar and two menus that contain some menu items. Clicking on them doesn't do much until we add some commands. That's what we will do next. Perform the following actions, above the code for the creation of the menu bar:

1. Open `GUI_menubar_help.py` and save it as `GUI_menubar_exit_quit.py`.
2. Create a `quit` function:

```
def _quit():
    win.quit()
    win.destroy()
    exit()
```

3. Next, we'll bind the **File** | **Exit** menu item to this function by adding the following command to the menu item:

```
file_menu.add_command(label="Exit", command=_quit)
```

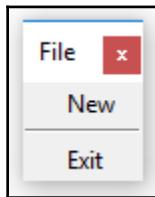
The preceding instructions produce the following code, which can be found in the `GUI_menubar_exit_quit.py` file:

```
# Exit GUI cleanly
def _quit():
    win.quit()
    win.destroy()
    exit()

# Creating a Menu Bar
menu_bar = Menu(win)
win.config(menu=menu_bar)

# Add menu items
file_menu = Menu(menu_bar, tearoff=0)
file_menu.add_command(label="New")
file_menu.add_separator()
file_menu.add_command(label="Exit", command=_quit)
menu_bar.add_cascade(label="File", menu=file_menu)
```

- Run the code and click the **Exit** menu item. The following GUI shows the output of the code we run:



When we click the **Exit** menu item, our application will indeed exit.

Now let's go behind the scenes to understand the code better.

How it works...

First, we call the `tkinter` constructor of the `Menu` class. Then, we assign the newly created menu to our main GUI window. This, in fact, becomes the menu bar. We save a reference to it in the instance variable named `menu_bar`.

Next, we create a menu and add two menu items to the menu. The `add_cascade()` method aligns the menu items one below the other, in a vertical layout.

Then, we add a separator line between the two menu items. This is generally used to group related menu items (hence the name).

Finally, we disable the `tearoff` dashed line to make our menu look much better.



Without disabling this default feature, the user can tear off the menu from the main window. I find this capability of little value. Feel free to play around with it by double-clicking the dashed line (before disabling this feature). If you are using a Mac, this feature might not be enabled; if so, you don't have to worry about it.

We then add a second menu to the menu bar. We can keep on adding menus using this technique.

Next, we create a function to quit our GUI application cleanly. How we quit a running Python application is the recommended Pythonic way to end the main event loop.

We bind the function we created to the menu item, which is the standard way of binding a function to a menu item, using `command` attribute of `tkinter`. Whenever we want our menu items to actually do something, we have to bind each of them to a function.



We are using a recommended Python naming convention by preceding our quit function with one single underscore. This indicates that this is a private function that can't be called by the clients of our code.

There's more...

We will add the **Help | About** functionality in Chapter 3, *Look and Feel Customization*, which introduces message boxes and much more.

We've successfully learned how to create menu bars. Now let's move on to the next recipe.

Creating tabbed widgets

In this recipe, we will create tabbed widgets to further organize our expanding GUI written in `tkinter`.

Getting ready

To improve our Python GUI using tabs, we will start at the beginning, using as little code as possible. In this recipe, we will create a simple GUI and then add widgets from the previous recipes, placing them in this new tabbed layout.

How to do it...

Follow these steps to create *Tab* controls, which in `tkinter` are called `Notebook`:

1. Create a new Python module and name it `GUI_tabbed.py`.
2. At the top of the module, import `tkinter`:

```
import tkinter as tk
from tkinter import ttk
```

3. Create an instance of the Tk class:

```
win = tk.Tk()
```

4. Add a title via the title attribute:

```
win.title ("Python GUI")
```

5. Create tabControl using the ttk Notebook:

```
tabControl = ttk.Notebook(win)
```

6. Add the tab to tabControl:

```
tabControl.add(tab1, text='Tab 1')
```

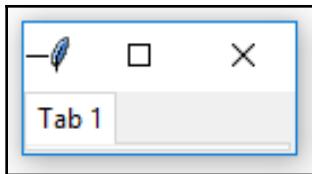
7. Use pack to make the control visible inside the GUI:

```
tabControl.pack(expand=1, fill="both")
```

The preceding instructions produce the following code, which can be found in the `GUI_tabbed.py` file:

```
6⑩ =====
7 # imports
8 =====
9⑩ import tkinter as tk
10 from tkinter import ttk
11
12 win = tk.Tk()                      # Create instance
13 win.title("Python GUI")             # Add a title
14 tabControl = ttk.Notebook(win)       # Create Tab Control
15 tab1 = ttk.Frame(tabControl)         # Create a tab
16 tabControl.add(tab1, text='Tab 1')    # Add the tab
17 tabControl.pack(expand=1, fill="both") # Pack to make visible
18
19⑩ =====
20 # Start GUI
21 =====
22 win.mainloop()
```

8. Run the preceding code. The following screenshot shows the GUI after running the code:



This widget adds another very powerful tool to our GUI design toolkit. It comes with its own limitations, all of which can be seen in this recipe (for example, we can neither reposition the GUI nor does it show the entire GUI title).



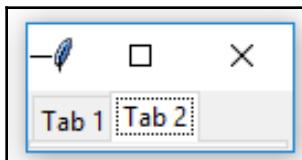
While we used the grid layout manager for simpler GUIs in the previous recipes, we can use a simpler layout manager: `pack` is one of them.

In the preceding code, we pack the `tabControl` and `ttk.Notebook` widgets into the main GUI form, expanding the notebook-tabbed control to fill in all the sides. We can add a second tab to our control and click between them by performing the following steps:

1. Open `GUI_tabbed.py` and save it as `GUI_tabbed_two.py`.
2. Add a second tab:

```
tab2 = ttk.Frame(tabControl) # Add a second tab
tabControl.add(tab2, text='Tab 2') # Add second tab
```

3. Run the preceding code. In the following screenshot, we have two tabs. Click on **Tab 2** to give it focus:



We would really like to see our window's title; to do this, we have to add a widget to one of our tabs. The widget has to be wide enough to expand our GUI dynamically so as to display our window title. Follow these steps to do so:

1. Open `GUI_tabbed_two.py` and save it as `GUI_tabbed_two_mighty.py`.

2. Add a LabelFrame and a Label:

```
# LabelFrame using tab1 as the parent
mighty = ttk.LabelFrame(tab1, text=' Mighty Python ')
mighty.grid(column=0, row=0, padx=8, pady=4)

# Label using mighty as the parent
a_label = ttk.Label(mighty, text="Enter a name:")
a_label.grid(column=0, row=0, sticky='W')
```

3. Run the preceding code. As shown in the following screenshot, we have **Mighty Python** inside **Tab 1**. This expands our GUI, but the added widgets aren't large enough to make the GUI title visible:



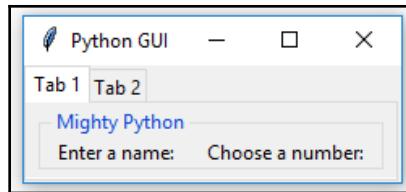
After adding a second label plus some spacing around them, we stretch the layout enough so we can see our GUI title again:

1. Open `GUI_tabbed_two_mighty.py` and save it as `GUI_tabbed_two_mighty_labels.py`.
2. Add a second label and spacing via a loop:

```
# Add another label
ttk.Label(mighty, text="Choose a number:").grid(column=1, row=0)

# Add some space around each label
for child in mighty.winfo_children():
    child.grid_configure(padx=8)
```

3. Run the preceding code. The following screenshot shows the output from running this code, which can also be found in the `GUI_tabbed_two_mighty_labels.py` file:

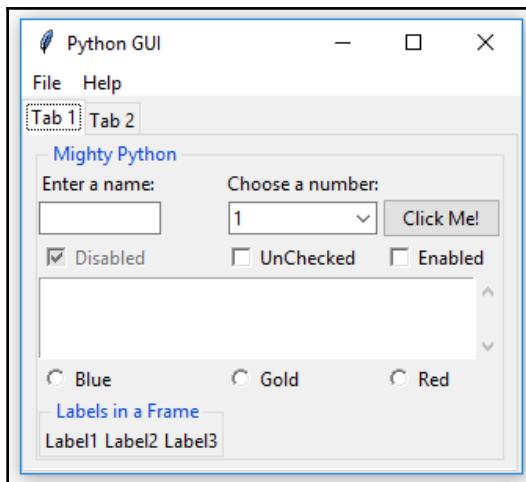


We can keep placing all the widgets we have created so far into our newly created tab controls.



You can download the code from <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition>. Try to create the tabbed GUI yourself. We have created and aligned all of the widgets in the previous recipes, but without placing them onto two different tabs.

Look at the `GUI_tabbed_all_widgets.py` file:



As you can see, all the widgets reside inside **Tab 1**. Let's move some of them to **Tab 2**:

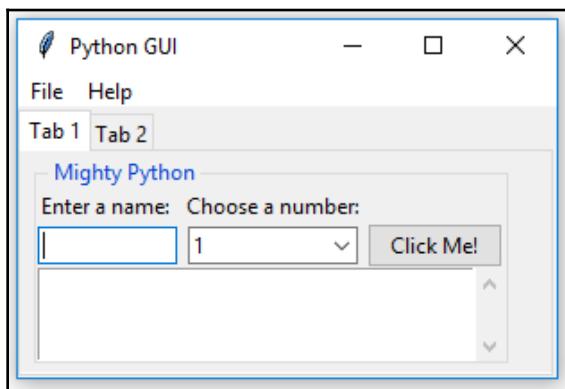
1. Create a second `LabelFrame`, which will be the container of the widgets we will be relocating to **Tab 2**:

```
mighty2 = ttk.LabelFrame(tab2, text=' The Snake ')
mighty2.grid(column=0, row=0, padx=8, pady=4)
```

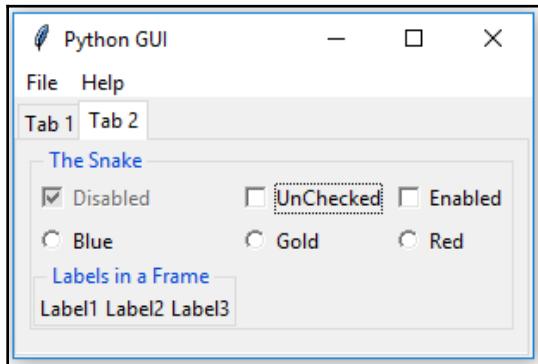
2. Next, we move the Check and Radio buttons to **Tab 2** by specifying the new parent container, which is a new variable that we name `mighty2`. The following is an example that we will apply to all the controls that relocate to **Tab 2**:

```
chVarDis = tk.IntVar()  
check 1 = tk.Checkbutton(mighty2, text="Disabled",  
variable=chVarDis,  
state='disabled')
```

3. Run the `GUI_tabbed_all_widgets_both_tabs.py` file. The following screenshot shows the output we receive after running the preceding code:



We can now click on **Tab 2** and see our relocated widgets:



After running the preceding code, our GUI looks different. **Tab 1** has fewer widgets than it had before when it contained all of our previously created widgets.

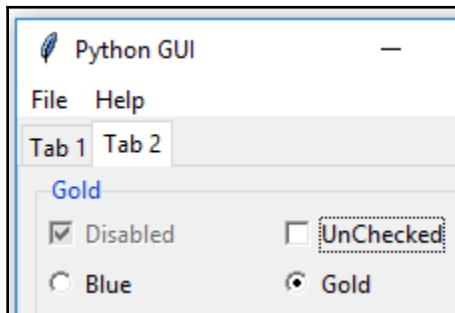
Clicking the relocated Radiobutton no longer has any effect, so we will change their actions to renaming the text attribute, from the title of the LabelFrame widget to the name the Radiobuttons display. When we click the **Gold** Radiobutton, we no longer set the background of the frame to the color gold. Instead, we replace the LabelFrame text title. Python's **The Snake** now becomes **Gold**:

```
def radCall():
    radSel=radVar.get()
    if radSel == 0: mighty2.configure(text ='Blue')
    if radSel == 1: mighty2.configure(text ='Gold')
    if radSel == 0: mighty2.configure(text ='Red')
```

4. Now selecting any of the RadioButton widgets will change the name of the LabelFrame.
5. Run the `GUI_tabbed_all_widgets_both_tabs_radio.py` file. The following screenshot shows the output of running the code in this file:



Notice how the label frame is now titled **Blue**. Clicking on the **Gold** radio button changes this title to **Gold**, as shown in the following screenshot:



Now let's go behind the scenes to understand the code better.

How it works...

On executing the code to create **Tab 1**, it is created but without any information in it. We then created a second tab, **Tab 2**. After creating the second tab, we moved some of the widgets that originally resided in **Tab 1** to **Tab 2**. Adding tabs is another excellent way to organize our ever-increasing GUI. This is a nice way to handle the complexity of our GUI design. We can arrange widgets in groups, where they naturally belong, and free our users from clutter by using tabs.



In `tkinter`, creating tabs is done via the `Notebook` widget, which is the tool that allows us to add tabbed controls. The `tkinter notebook` widget, like so many other widgets, comes with additional properties that we can use and configure. An excellent place to start exploring the additional capabilities of the `tkinter` widgets at our disposal is the official website: <https://docs.python.org/3.1/library/tkinter.ttk.html#notebook>.

We've successfully learned how to create tabbed widgets. Now let's move on to the next recipe.

Using the grid layout manager

The grid layout manager is one of the most useful layout tools at our disposal. While layout tools such as pack are simple and easy to use, grid gives us a lot of control over our layout – especially when we combine grid with embedded frames.

We have already used it in many recipes, for example, because it is just so powerful.

Getting ready...

In this recipe, we will review some grid layout manager techniques. We have already used them, but we will explore them in more detail here.

How to do it...

In this chapter, we have created rows and columns, which is the database approach to GUI design (MS Excel does the same). We hard-coded the first rows. However, if we forget to specify where we want the next row to reside, tkinter fills this in without us even noticing.

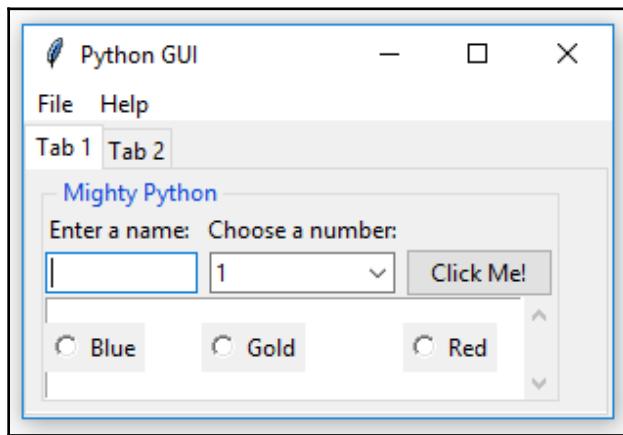
To observe this, let's take the code from a recipe we previously worked on:

1. Open `GUI_tabbed_all_widgets_both_tabs_radio.py`.
2. Comment out the `scr.grid` line, as follows:

```
# Using a scrolled Text control
scrol_w  = 30
scrol_h  = 3
scr = scrolledtext.ScrolledText(mighty, width=scrol_w, height=scrol_h, wrap=tk.WORD)
# scr.grid(column=0, row=2, sticky='WE', columnspan=3)
scr.grid(column=0, sticky='WE', columnspan=3)                                # row not specified
```

tkinter automatically adds the missing row to where we didn't specify any particular row.

3. Run the code and notice how our radio buttons suddenly ended up in the middle of the Text widget!



Now let's go behind the scenes to understand the code better.

How it works...

We laid out the `Entry` widgets on row 1. However, we forgot to specify the row for our `ScrolledText` widget, which we reference via the `scr` variable. Then, we added the `Radiobutton` widgets we want to be laid out in row 3.

This works nicely because `tkinter` automatically incremented the row position for our `ScrolledText` widget so it used the next highest row number, which was row 2.

Looking at our code and not realizing that we forgot to explicitly position our `ScrolledText` widget to row 2, we might think nothing resides there.

Due to this, we might try the following. If we set the `curRad` variable to use row 2, we might get an unpleasant surprise, as shown in the final screenshot in the *How to do it...* section of this recipe.

Now let's go behind the scenes to understand the code better.

Note how our row of `RadioButton`(s) suddenly ended up in the middle of our `ScrolledText` widget! This is definitely not what we intended our GUI to look like!



If we forget to explicitly specify the row number, by default, `tkinter` will use the next available row.

We also used the `columnspan` property to make sure our widgets did not get limited to just one column, as shown in the following screenshot:

```
scr = scrolledtext.ScrolledText(mighty, width=scrol_w, height=scrol_h, wrap=tk.WORD)
scr.grid(column=0, row=5, sticky='WE', columnspan=3)
```

The preceding screenshot shows how we made sure that our `ScrolledText` widget spans all the columns in our GUI.

3

Look and Feel Customization

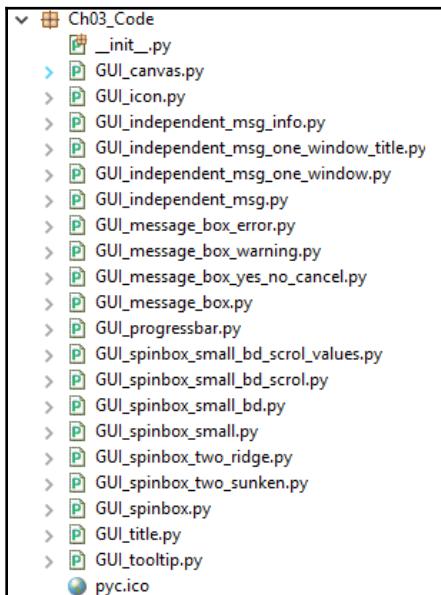
In this chapter, we will customize some of the widgets in our GUI by changing some of their attributes. We will also introduce a few new widgets that `tkinter` offers us.

In the *Creating tooltips using Python* recipe, we will create a `ToolTip` OOP-style class, which will be a part of the single Python module that we have been using until now.

You will learn how to create different message boxes, change the GUI window title, and much more. We will be using a spin box control to learn how to apply different styles.

Look and feel customization is a very important part of GUI design because it makes our GUI look professional.

Here is the overview of the Python modules for this chapter:



In this chapter, we will customize our GUI using Python 3.7 and above. We will cover the following recipes:

- Creating message boxes – the information, warning, and error
- How to create independent message boxes
- How to create the title of a tkinter window form
- Changing the icon of the main root window
- Using a spin box control
- Applying relief – sunken and raised appearance of widgets
- Creating tooltips using Python
- Adding Progressbar to the GUI
- How to use the canvas widget

Creating message boxes – information, warning, and error

A message box is a pop-up window that gives feedback to the user. It can be informational, hinting at potential problems, as well as catastrophic errors.

Using Python to create message boxes is very easy.

Getting ready

We will add functionality to the **Help | About** menu item we created in [Chapter 2, Layout Management](#), in the *Creating tabbed widgets* recipe.

The code is from `GUI_tabbed_all_widgets_both_tabs.py`. The typical feedback to the user when clicking the **Help | About** menu in most applications is informational. We'll start with this information and then vary the design pattern to show warnings and errors.

How to do it...

Here are the steps to follow to create a message box in Python:

1. Open `GUI_tabbed_all_widgets_both_tabs.py` from Chapter 2, *Layout Management*, and save the module as `GUI_message_box.py`.
2. Add the following line of code to the top of the module where the import statements live:

```
from tkinter import messagebox as msg
```

3. Next, create a callback function that will display a message box. We have to place the code of the callback above the code where we attach the callback to the menu item, because this is still procedural and not OOP code.

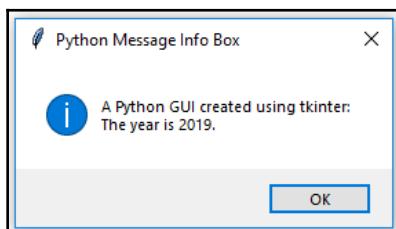
Add the following code just above the lines where we create the help menu:

```
def _msgBox():  
    msg.showinfo('Python Message Info Box', 'A Python GUI created  
    using tkinter:\nThe year is 2019.')
```

The preceding instructions produce the following code, `GUI_message_box.py`:

```
# Display a Message Box  
def _msgBox():  
    msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2019.')  
  
# Add another Menu to the Menu Bar and an item  
help_menu = Menu(menu_bar, tearoff=0)  
help_menu.add_command(label="About", command=_msgBox) # display messagebox when clicked  
menu_bar.add_cascade(label="Help", menu=help_menu)
```

4. Run the code. Clicking **Help | About** now causes the following pop-up window to appear:



Let's transform this code into a warning message box pop-up window instead:

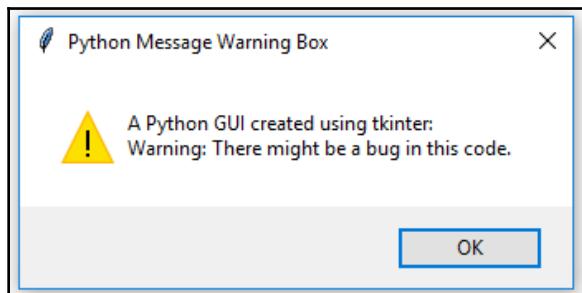
1. Open `GUI_message_box.py` and save the module as `GUI_message_box_warning.py`.
 2. Comment out the `msg.showinfo` line.
 3. Replace the information box code with warning box code:

```
msg.showwarning('Python Message Warning Box', 'A Python GUI created  
using tkinter:' '\nWarning: There might be a bug in this code.')
```

The preceding instructions produce the following code, `GUI_message_box_warning.py`:

```
# Display a Message Box
def _msgBox():
#     msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:'
#                 '\nThe year is 2019.')
    msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:'
                   '\nWarning: There might be a bug in this code.')
```

4. Running the preceding code will now result in the following slightly modified message box:



Displaying an error message box is simple and usually warns the user of a serious problem. As we did in the previous code snippet, comment out the previous line and add the following code, as we have done here:

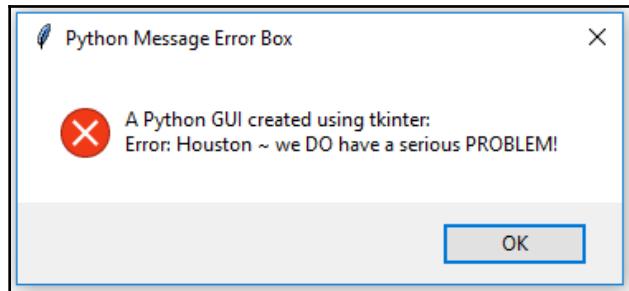
1. Open `GUI_message_box_warning.py` and save the module as `GUI_message_box_error.py`.
2. Replace the warning box code with error box code:

```
msg.showerror('Python Message Error Box', 'A Python GUI created  
using tkinter:  
\nError: Houston ~ we DO have a serious PROBLEM!')
```

The preceding instructions produce the following code:

```
# Display a Message Box  
def _msgBox():  
#     msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2019.')  
#     msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:  
#\nWarning: There might be a bug in this code.')  
    msg.showerror('Python Message Error Box', 'A Python GUI created using tkinter:  
#\nError: Houston ~ we DO have a serious PROBLEM!')
```

3. Run the `GUI_message_box_error.py` file. The error message looks like this:



There are different message boxes that display more than one **OK** button, and we can program our responses according to the user's selection.

The following is a simple example that illustrates this technique:

1. Open `GUI_message_box_error.py` and save the module as `GUI_message_box_yes_no_cancel.py`.

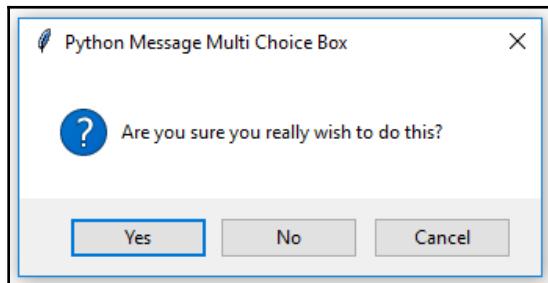
2. Replace the error box with a yes_no_cancel box:

```
answer = msg.askyesnocancel("Python Message Multi Choice Box", "Are  
you sure you really wish to do this?")
```

The preceding instructions produce the following code:

```
# Display a Message Box  
def _msgBox():  
#     msg.showinfo('Python Message Info Box', 'A Python GUI created using tkinter:\nThe year is 2019.')  
#     msg.showwarning('Python Message Warning Box', 'A Python GUI created using tkinter:\nWarning: There might be a bug in this code.')  
#     msg.showerror('Python Message Error Box', 'A Python GUI created using tkinter:\nError: Houston ~ we DO have a serious PROBLEM!')  
    answer = msg.askyesnocancel("Python Message Multi Choice Box", "Are you sure you really wish to do this?")  
    print(answer)
```

3. Run the `GUI_message_box_yes_no_cancel.py` file. Running this GUI code results in a popup whose user response can be used to branch on the answer of this event-driven GUI loop, by saving it in the `answer` variable:



The console output using Eclipse shows that clicking the **Yes** button results in the Boolean value of **True** being assigned to the `answer` variable:

```
Console X  
<terminated> GUI_message_box_yes_no_cancel.py [C:\Python37\python.exe]  
False  
None  
True
```

For example, we could use the following code:

```
If answer == True:  
    <do something>
```

Clicking **No** returns `False` and **Cancel** returns `None`.

Now, let's go behind the scenes to understand the code better.

How it works...

We added another callback function to all of our `GUI_message_box` Python modules, `_msgBox()`, and attached it to the **Help** menu `command` attribute to handle click events. Now, when we click the **Help | About** menu, an action takes place. We are creating and displaying the most common pop-up message box dialogs. They are modal, so the user can't use the GUI until they click the **OK** button.

In the first example, we display an information box, as can be seen by the icon to its left. Next, we create warning and error message boxes, which automatically change the icon associated with the popup. All we have to do is specify which message box we want to display.

The `askyesnocancel` message box returns a different value depending on which button the user clicked. We can capture the answer in a variable and write different code according to which answer was selected.

We've successfully learned how to create message boxes. Now, let's move on to the next recipe.

How to create independent message boxes

In this recipe, we will create our `tkinter` message boxes as standalone top-level GUI windows.

You will first notice that, by doing so, we end up with an extra window, so we will explore ways to hide this window.

In the previous recipe, we invoked `tkinter` message boxes via our **Help | About** menu from our main GUI form.

So, why would we wish to create an independent message box?

One reason is that we might customize our message boxes and reuse them in several of our GUIs. Instead of having to copy and paste the same code into every Python GUI we design, we can factor it out of our main GUI code. This creates a small reusable component, which we can then import into different Python GUIs.

Getting ready

We have already created the title of a message box in the previous recipe, *Creating message boxes - information, warning, and error*. We will not reuse the code from the previous recipe, but build a new GUI using very few lines of Python code.

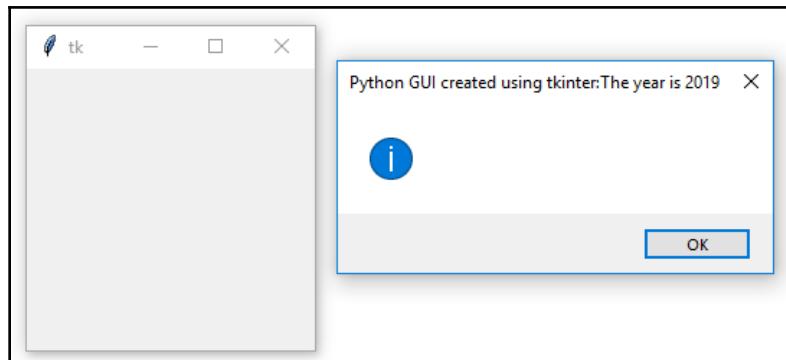
How to do it...

We can create a simple message box as follows:

1. Create a new module and save it as `GUI_independent_msg.py`.
2. Add the following two lines of code, which is all that is required:

```
from tkinter import messagebox as msg  
msg.showinfo('Python GUI created using tkinter:\nThe year is 2019')
```

3. Run the `GUI_independent_msg.py` file. This will result in the following two windows:



This does not look like what we had in mind. Now, we have two windows, one undesired and the second with its text displayed as its title.

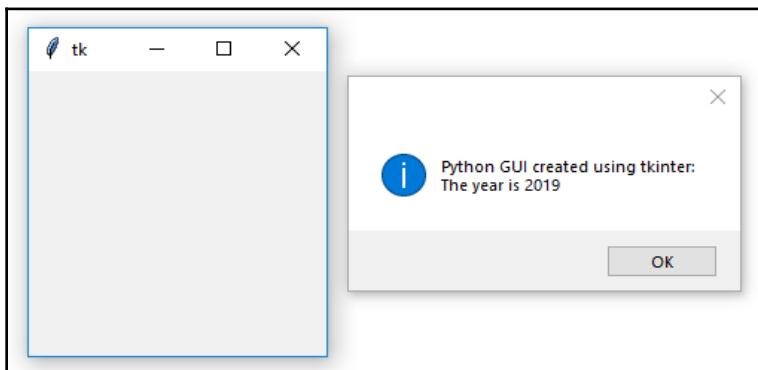
Oops!

Let's solve this now. We can change the Python code by adding a single or double quote followed by a comma:

1. Open `GUI_independent_msg.py` and save the module as `GUI_independent_msg_info.py`.
2. Create an empty title:

```
from tkinter import messagebox as msg
msg.showinfo('', 'Python GUI created using tkinter:\nThe year is
2019')
```

3. Run the `GUI_independent_msg_info.py` file. Now, we do not have a title but our text ended up inside the popup, as we had intended:



The first parameter is the title and the second is the text displayed in the pop-up message box. By adding an empty pair of single or double quotes followed by a comma, we can move our text from the title into the pop-up message box.

We still need a title, and we definitely want to get rid of this unnecessary second window. The second window is caused by a Windows event loop. We can get rid of it by suppressing it.

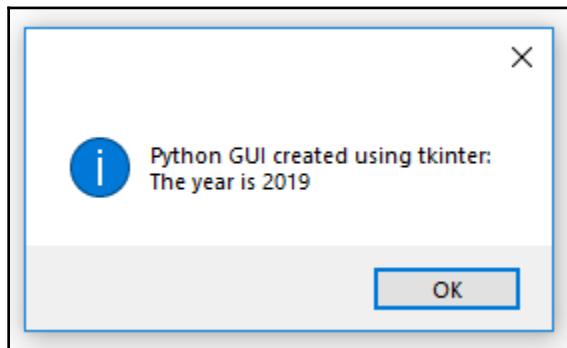
Add the following code:

1. Open `GUI_independent_msg_info.py` and save the module as `GUI_independent_msg_one_window.py`.
2. Import Tk create an instance of the Tk class, and call the withdraw method:

```
from tkinter import Tk  
root = Tk()  
root.withdraw()
```

Now, we have only one window. The `withdraw()` method removes the debug window that we are not interested in having floating around.

3. Run the code. This will result in the following window:



In order to add a title, all we have to do is place string into our empty first argument.

For example, consider the following code snippet:

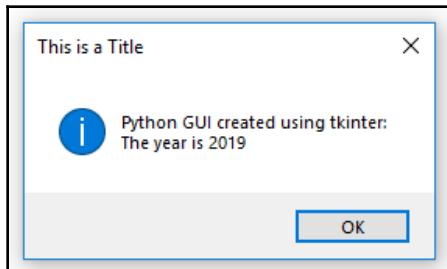
1. Open `GUI_independent_msg_one_window.py` and save the module as `GUI_independent_msg_one_window_title.py`.
2. Give it a title by adding some words into the first argument position:

```
msg.showinfo('This is a Title', 'Python GUI created using  
tkinter:\n\nThe year is 2019')
```

The preceding instructions produce the following code:

```
from tkinter import messagebox as msg
from tkinter import Tk
root = Tk()
root.withdraw()
msg.showinfo('This is a Title', 'Python GUI created using tkinter:\nThe year is 2019')
```

3. Run the `GUI_independent_msg_one_window_title.py` file. Now, our dialog has a title, as shown in the following screenshot:



Now, let's go behind the scenes to understand the code better.

How it works...

We pass more arguments into the `tkinter` constructor of the message box to add a title to the window form and display the text in the message box instead of displaying it as its title. This happens due to the position of the arguments we pass. If we leave out an empty quote or a double quote, then the message box widget takes the first position of the arguments as the title, not the text to be displayed within the message box. By passing an empty quote followed by a comma, we change where the message box displays the text we pass into the function.

We suppress the second pop-up window, which automatically gets created by the `tkinter` message box widget, by calling the `withdraw()` method on our main root window.

By adding some words into the previously empty string, we give our message box a title. This shows that the different message boxes, in addition to the main message they are displaying, have their own custom title. This can be useful to relate several different message boxes to the same functionality.

We've successfully learned how to create independent message boxes. Now, let's move on to the next recipe.

How to create the title of a tkinter window form

The principle of changing the title of a `tkinter` main root window is the same as we discussed in the previous recipe: *How to create independent message boxes*. We just pass in a string as the first argument to the constructor of the widget.

Getting ready

Instead of a pop-up dialog window, we create the main root window and give it a title.

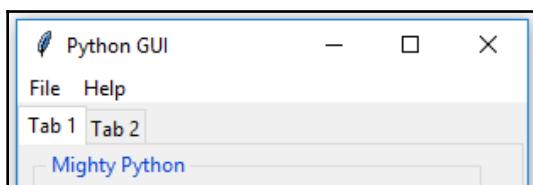
How to do it...

The following code creates the main window and adds a title to it. We have already done this in the previous recipes; for example, in the *Creating tabbed widgets* recipe, in Chapter 2, *Layout Management*. Here, we just focus on this aspect of our GUI:

1. Open `GUI_tabbed_all_widgets_both_tabs.py` and save the module as `GUI_title.py`.
2. Give the main window a title:

```
import tkinter as tk
win = tk.Tk()                      # Create instance
win.title("Python GUI")             # Add a title
```

3. Run the `GUI_title.py` file. This will result in the following two tabs:



Now, let's go behind the scenes to understand the code better.

How it works...

This gives a title to the main root window by using the built-in the `title` attribute of `tkinter`. After we create a `Tk()` instance, we can use all the built-in `tkinter` attributes to customize our GUI.

We've successfully learned how to create a title for a `tkinter` window form. Now, let's move on to the next recipe.

Changing the icon of the main root window

One way to customize our GUI is to give it an icon different from the default icon that ships out of the box with `tkinter`. Here is how we do this.

Getting ready

We are improving our GUI from the *Creating tabbed widgets* recipe in chapter 2, *Layout Management*. We will use an icon that ships with Python, but you can use any icon you find useful. Make sure you have the full path to where the icon lives in your code, or you might get errors.

How to do it...

For this example, I have copied the icon from where I installed Python 3.7 to the same folder where the code lives. The following screenshot shows the icon that we will be using:

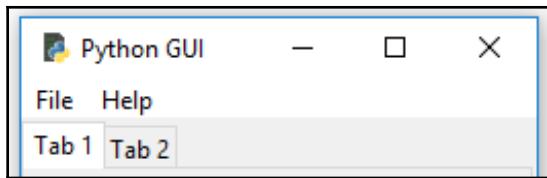


In order to use this or another icon file, perform the following steps:

1. Open `GUI_title.py` and save the module as `GUI_icon.py`.
2. Place the following code above the main event loop:

```
# Change the main windows icon  
win.iconbitmap('pyc.ico')
```

3. Run the `GUI_icon.py` file. Observe how the feather default icon in the top-left corner of the GUI changed:



Now, let's go behind the scenes to understand the code better.

How it works...

This is another attribute that ships with `tkinter`, which ships with Python 3.7 and above. We use the `iconbitmap` attribute to change the icon of our main root window form, by passing in a relative path to an icon. This overrides the default icon of `tkinter`, replacing it with our icon of choice.



If the icon is located in the same folder where the Python module is located, we can simply refer to the icon by its name without passing in the full path to the icon location.

We've successfully learned how to change the icon of the main root window. Now, let's move on to the next recipe.

Using a spin box control

In this recipe, we will use a `Spinbox` widget, and we will also bind the `Enter` key on the keyboard to one of our widgets. The `Spinbox` widget is a one-line widget, like the `Entry` widget, with the additional capability to restrict the values it will display. It also has some small up/down arrows to scroll up and down between the values.

Getting ready

We will use our tabbed GUI, from the *How to create the title of a tkinter window form* recipe, and add a `Spinbox` widget above the `ScrolledText` control. This simply requires us to increment the `ScrolledText` row value by one and insert our new `Spinbox` control in the row above the `Entry` widget.

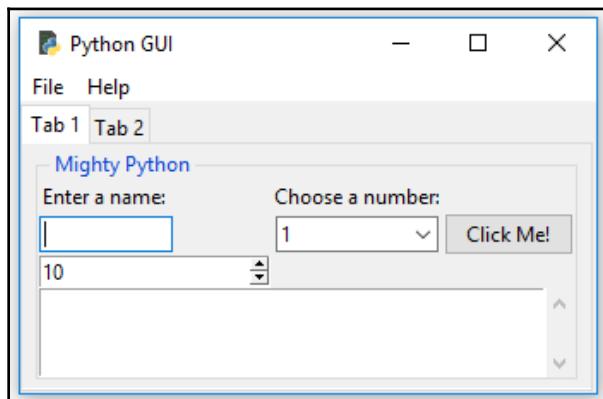
How to do it...

First, we add the `Spinbox` control by performing the following instructions:

1. Open `GUI_title.py` and save the module as `GUI_spinbox.py`.
2. Place the following code above the `ScrolledText` widget:

```
# Adding a Spinbox widget
spin = Spinbox(mighty, from_=0, to=10)
spin.grid(column=0, row=2)
```

3. Run the code. This will modify our GUI as follows:

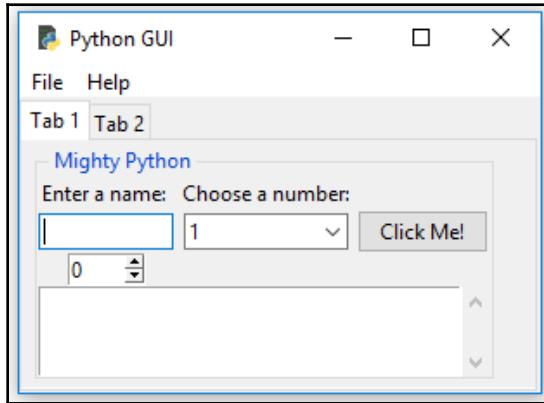


Next, we will reduce the size of the Spinbox widget:

1. Open `GUI_spinbox.py` and save the module as `GUI_spinbox_small.py`.
2. Add a `width` attribute when creating the Spinbox widget:

```
spin = Spinbox(mighty, from_=0, to=10, width=5)
```

3. Running the preceding code results in the following GUI:

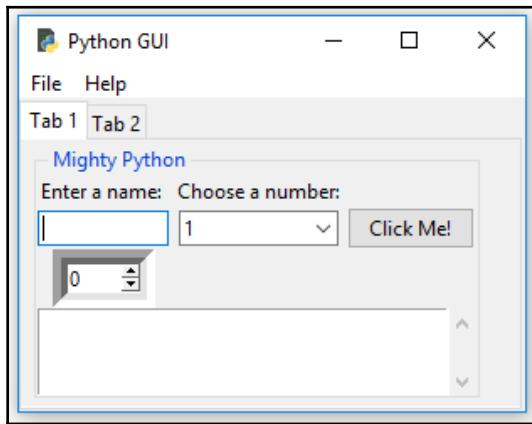


Next, we add another attribute to customize our widget further; `bd` is short-hand notation for the `borderwidth` attribute, and changes the width of the border surrounding the spin box:

1. Open `GUI_spinbox_small.py` and save the module as `GUI_spinbox_small_bd.py`.
2. Add a `bd` attribute, giving it a size of 8:

```
spin = Spinbox(mighty, from_=0, to=10, width=5, bd=8)
```

3. Running the preceding code results in the following GUI:



Next, we add functionality to the widget by creating a callback and linking it to the control.

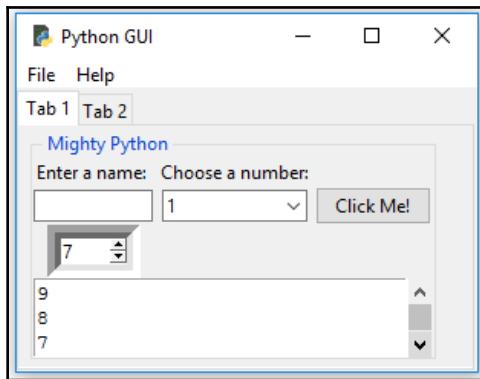
The following steps show how to print the selection of the Spinbox widget into ScrolledText as well as onto stdout. The variable named `scrol` is our reference to the ScrolledText widget:

1. Open `GUI_spinbox_small_bd.py` and save the module as `GUI_spinbox_small_bd_scrol.py`.
2. Write a callback function right above the creation of the Spinbox widget and assign it to the command attribute of the Spinbox: widget:

```
# Spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scrol.insert(tk.INSERT, value + '\n')    # <-- add a newline

spin = Spinbox(mighty, from_=0, to=10, width=5, bd=8,
              command=_spin)                      # <-- command=_spin
```

3. Running the `GUI_spinbox_small_bd_scrol.py` file results in the following GUI when clicking the Spinbox arrows:

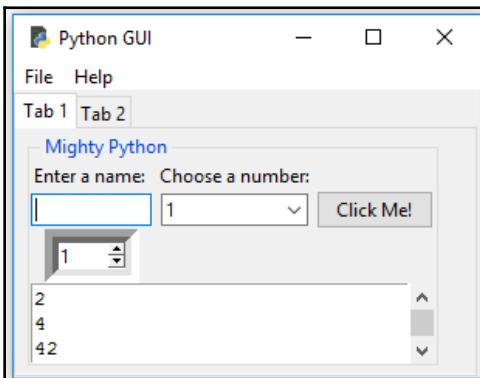


Instead of using a range, we can also specify a set of values by performing the following instructions:

1. Open `GUI_spinbox_small_bd_scrol.py` and save the module as `GUI_spinbox_small_bd_scrol_values.py`.
2. Add the `values` attribute, replacing `from_=0, to=10`, and assign it a tuple of numbers during the creation of the Spinbox widget:

```
# Adding a Spinbox widget using a set of values
spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5, bd=8,
               command=_spin)
spin.grid(column=0, row=2)
```

3. Run the code. This will create the following GUI output:



Now, let's go behind the scenes to understand the code better.

How it works...

Note how, in the first Python module, `GUI_spinbox.py`, our new Spinbox control defaulted to a width of 20, pushing out the column width of all controls in this column. This is not what we want. We gave the widget a range from 0 to 10.

In the second Python module, `GUI_spinbox_small.py`, we reduced the width of the Spinbox control, which aligned it in the center of the column.

In the third Python module, `GUI_spinbox_small_bd.py`, we added the borderwidth attribute of the Spinbox, which automatically made the entire Spinbox appear no longer flat, but three-dimensional.

In the fourth Python module, `GUI_spinbox_small_bd_scrol.py`, we added a callback function to display the number chosen in the `ScrolledText` widget and also print it to the standard out stream. We added `\n` to insert the values on new lines within the callback function, `def _spin()`.

Notice how the default value does not get printed. It is only when we click the control that the callback function gets called. By clicking the down arrow with a default of 0, we can print the 0 value.

Lastly, in `GUI_spinbox_small_bd_scrol_values.py`, we restricted the values available to a hardcoded set. This could also be read in the form of a data source (for example, a text or XML file).

We've successfully learned how to use a spin box control. Now, let's move on to the next recipe.

Applying relief – the sunken and raised appearance of widgets

We can control the appearance of our Spinbox widgets by using an attribute that makes them appear in different formats, such as sunken or raised. This attribute is the `relief` attribute.

Getting ready

We will add one more Spinbox control to demonstrate the available appearances of widgets, using the `relief` attribute of the Spinbox control.

How to do it...

While we are creating the second Spinbox, let's also increase `borderwidth` to distinguish our second Spinbox from the first Spinbox:

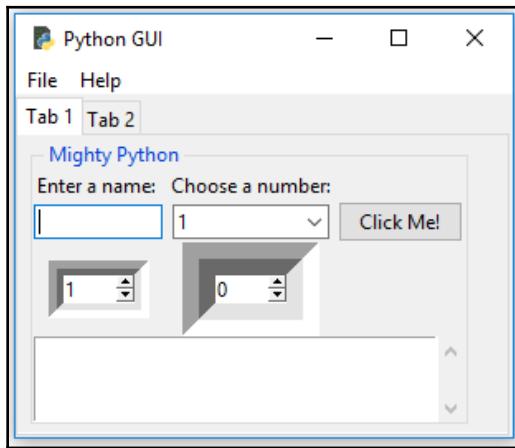
1. Open `GUI_spinbox_small_bd_scrol_values.py` and save the module as `GUI_spinbox_two_sunken.py`.
2. Add a second Spinbox just below the first Spinbox and set `bd=20`:

```
# Adding a second Spinbox widget
spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=20,
                command=_spin2)                         # <-- new function
spin2.grid(column=1, row=2)
```

3. We will also create a new callback function for the `command` attribute, `_spin2`. Place this function *above* the code just shown, where we create the second Spinbox:

```
# Spinbox2 callback function
def _spin2():
    value = spin2.get()
    print(value)
    scrol.insert(tk.INSERT, value + '\n')
    # <-- write to same ScrolledText
```

4. Run the code. This will create the following GUI output:



Our two spin boxes look different but this is only because of the difference in the `borderwidth (bd)` we specified. Both widgets look three-dimensional, and this is much more visible in the second Spinbox that we have added.

They actually both have a `relief` style even though we did not specify the `relief` attribute when we created the spin boxes.

When not specified, the `relief` style defaults to `SUNKEN`.

Here are the available `relief` attribute options that can be set:

- `tk.SUNKEN`
- `tk.RAISED`
- `tk.FLAT`
- `tk.GROOVE`
- `tk.RIDGE`



We imported `tkinter` as `tk`. This is why we can call the `relief` attribute as `tk.SUNKEN`, and so on.

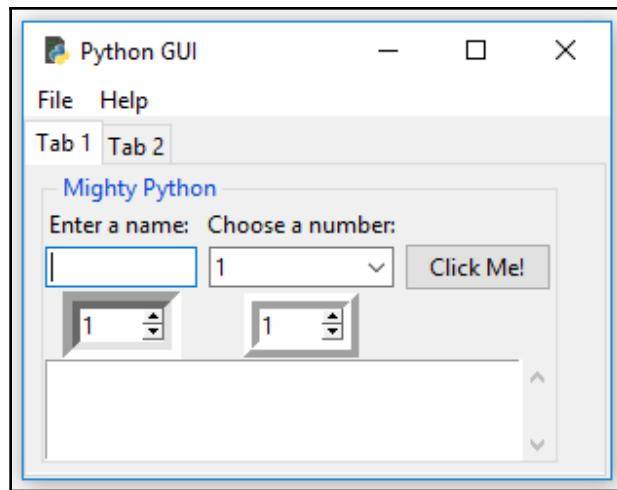
By assigning the different available options to the `relief` attribute , we can create different appearances for this widget.

Assigning the `tk.RIDGE` relief and reducing the border width to the same value as our first Spinbox widget results in the following GUI:

1. Open `GUI_spinbox_two_sunken.py` and save the module as `GUI_spinbox_two_ridge.py`.
2. Set `relief` to `tk.RIDGE`:

```
spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9,  
command=_spin2, relief=tk.RIDGE)
```

3. Run the code. The following GUI is obtained after running the code:



Notice the difference in appearance of our second Spinbox widget, on the right.

Now, let's go behind the scenes to understand the code better.

How it works...

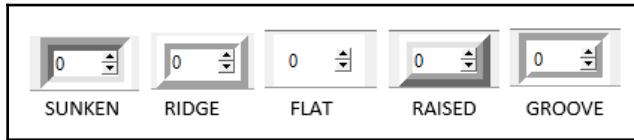
First, we created a second `Spinbox` aligned in the second column (`index == 1`). It defaults to `SUNKEN`, so it looks similar to our first `Spinbox`. We distinguished the two widgets by increasing the border width of the second control (the one on the right).

Next, we explicitly set the `relief` attribute of the `Spinbox` widget. We made `borderwidth` the same as our first `Spinbox` because, by giving it a different `relief`, the differences became visible without having to change any other attributes.

Here is an example of the different `relief` options, `GUI_spinbox_two_ridge.py`:

```
# Adding a second Spinbox widget displaying its relief options
# uncomment each next code line to see the different effects
spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.RIDGE)
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2)    # default value is: tk.SUNKEN
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.FLAT)
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.RAISED)
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.SUNKEN) # default
# spin2 = Spinbox(mighty, values=(0, 50, 100), width=5, bd=9, command=_spin2, relief=tk.GROOVE)
```

And here is a screenshot of what those relief attributes create:



We've successfully learned how to use and apply relief, sunken, and raised appearances to widgets. Now, let's move on to the next recipe.

Creating tooltips using Python

This recipe will show you how to create tooltips. When the user hovers the mouse over a widget, additional information will be available in the form of a tooltip.

We will code this additional information into our GUI.

Getting ready

We will be adding more useful functionality to our GUI. Surprisingly, adding a tooltip to our controls should be simple, but it is not as simple as we'd want it to be.

In order to achieve this desired functionality, we will place our tooltip code in its own OOP class.

How to do it...

These are the steps to create a tooltip:

1. Open `GUI_spinbox_small_bd_scrol_values.py` and save the module as `GUI_tooltip.py`.
2. Add the following class just below the `import` statements:

```
class ToolTip(object):
    def __init__(self, widget, tip_text=None):
        self.widget = widget
        self.tip_text = tip_text
        widget.bind('<Enter>', self.mouse_enter)
        widget.bind('<Leave>', self.mouse_leave)
```

3. Add two new methods to the class below `__init__`:

```
def mouse_enter(self, _event):
    self.show_tooltip()

def mouse_leave(self, _event):
    self.hide_tooltip()
```

4. Add another method below these two, and name the method `show_tooltip`:

```
def show_tooltip(self):
    if self.tip_window:
        x_left = self.widget.winfo_rootx()
        y_top = self.widget.winfo_rooty() - 18
        self.tip_window = tk.Toplevel(self.widget)
        self.tip_window.overrideredirect(True)
        self.tip_window.geometry("+%d+%d" % (x_left, y_top))
        label = tk.Label(self.tip_window, text=self.tip_text,
                         justify=tk.LEFT, background="#ffffe0", relief=tk.SOLID,
                         borderwidth=1, font=("tahoma", "8", "normal"))
        label.pack(ipadx=1)
```

5. Add another method below `show_tooltip`, and name it `hide_tooltip`:

```
def hide_tooltip(self):
    if self.tip_window:
        self.tip_window.destroy()
```

6. Below the class and below the code where we create the Spinbox widget, create an instance of the ToolTip class, passing in the Spinbox variable, spin:

```
# Adding a Spinbox widget
spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5, bd=9,
command=_spin)  spin.grid(column=0, row=2)

# Add a Tooltip to the Spinbox
ToolTip(spin, 'This is a Spin control')      # <-- add this code
```

7. Perform the same step for the ScrolledText widget just below the Spinbox widget:

```
scrol = scrolledtext.ScrolledText(mighty, width=scrol_w,
height=scrol_h, wrap=tk.WORD)
scrol.grid(column=0, row=3, sticky='WE', columnspan=3)

# Add a Tooltip to the ScrolledText widget
ToolTip(scrol, 'This is a ScrolledText widget') # <-- add this code
```

8. Run the code and hover the mouse over the ScrolledText widget:



Now, let's go behind the scenes to understand the code better.

How it works...

This is the beginning of OOP programming we'll do in this book. This might appear a little bit advanced, but do not worry; we will explain everything, and it does work.

We first created a new class and named it `ToolTip`. In the initializer method, `__init__`, we expect `widget` and `tip_text` to be passed in. We save these in instance variables, using the `self` keyword.

Next, we are bind the `Enter` and `Leave` mouse events to new methods that we create just below the initializer. These are being automatically called when we hover the mouse over a widget for which we have created a tooltip. These two methods call the next two methods of our class, which we create just below them.

The `show_tooltip` method checks whether a text was passed in during the creation of a `ToolTip` class instance and, if it was, we get the top-left coordinates of the widget, using `winfo_rootx` and `winfo_rooty`. These are `tkinter` built-in methods we can use.

For the `y_top` variable, we **subtract** 18, which positions the widget. This might seem counterintuitive, but the `tkinter` coordinate system starts with 0, 0 at the top-left corner of the screen, so subtracting from the `y` coordinate actually moves it up.

We then create a `TopLevel` window of `tkinter` for our tooltip. Setting `overrideredirect` (`True`) removes a toolbar that would otherwise be surrounding our tooltip, and we don't want that.

We use `geometry` to position our tooltip, and then we create a `Label` widget. We make our tooltip the parent of our label. We then use the tooltip text to be displayed inside the label.

We then pack the `Label` widget, which makes it visible.

In the `hide_tooltip` method, we check whether a tooltip has been created and, if so, we call the `destroy` method on it. Otherwise, whenever we hover the mouse over a widget and then move the mouse away from the widget, the tooltip will not go away.

With our `ToolTip` class code in place, we can now create tooltips for our widgets. We do this by creating an instance of the `ToolTip` class, passing in our `widget` variable and the text we wish to be displayed.

We do this for the `ScrolledText` and `Spinbox` widgets.

We've successfully learned how to create tooltips using Python. Now, let's move on to the next recipe.

Adding Progressbar to the GUI

In this recipe, we will add a `Progressbar` to our GUI. It is very easy to add a `ttk.Progressbar`, and we will demonstrate how to start and stop a `Progressbar`. This recipe will also show you how to delay the stopping of a `Progressbar`, and how to run it in a loop.

A `Progressbar` is typically used to show the current status of a long-running process.

Getting ready

We will add `Progressbar` to Tab 2 of the GUI that we developed in a previous recipe: *Using a spin box control*.

How to do it...

Here are the steps to create a `Progressbar` and some new Buttons that start and stop the `Progressbar`:

1. Open `GUI_spinbox_small_bd_scrol_values.py` and save the module as `GUI_progressbar.py`.
2. At the top of the module, add `sleep` to the imports:

```
from time import sleep # careful - this can freeze the GU
```

3. Add `Progressbar` below the code where we create the three `Radiobutton` widgets:

```
# Now we are creating all three Radiobutton widgets within one loop
for col in range(3):
    curRad = tk.Radiobutton(mighty2, text=colors[col],
                           variable=radVar, value=col, command=radCall)
    curRad.grid(column=col, row=1, sticky=tk.W) # row=6

# Add a Progressbar to Tab 2 # <--- add this code here
```

```
progress_bar = ttk.Progressbar(tab2, orient='horizontal',
                               length=286, mode='determinate')
progress_bar.grid(column=0, row=3, pady=2)
```

4. Next, we write a callback function to update Progressbar:

```
# update progressbar in callback loop
def run_progressbar():
    progress_bar["maximum"] = 100
    for i in range(101):
        sleep(0.05)
        progress_bar["value"] = i      # increment progressbar
        progress_bar.update()         # have to call update() in loop
    progress_bar["value"] = 0        # reset/clear progressbar
```

5. We then write the following three functions below the preceding code:

```
def start_progressbar():
    progress_bar.start()

def stop_progressbar():
    progress_bar.stop()

def progressbar_stop_after(wait_ms=1000):
    win.after(wait_ms, progress_bar.stop)
```

6. We will reuse buttons_frame and LabelFrame, but replace the labels with new code. Change the following code:

```
# PREVIOUS CODE -- REPLACE WITH BELOW CODE
# Create a container to hold labels
buttons_frame = ttk.LabelFrame(mighty2, text=' Labels in a Frame ')
buttons_frame.grid(column=0, row=7)

# NEW CODE
# Create a container to hold buttons
buttons_frame = ttk.LabelFrame(mighty2, text=' ProgressBar ')
buttons_frame.grid(column=0, row=2, sticky='W', columnspan=2)
```

7. Delete the previous labels that resided in buttons_frame:

```
# DELETE THE LABELS BELOW
# Place labels into the container element
ttk.Label(buttons_frame, text="Label1").grid(column=0, row=0,
sticky=tk.W)
ttk.Label(buttons_frame, text="Label2").grid(column=1, row=0,
sticky=tk.W)
ttk.Label(buttons_frame, text="Label3").grid(column=2, row=0,
sticky=tk.W)
```

8. Create four new buttons. buttons_frame is their parent:

```
# Add Buttons for Progressbar commands
ttk.Button(buttons_frame, text=" Run Progressbar ",
command=run_progressbar).grid(column=0, row=0, sticky='W')
ttk.Button(buttons_frame, text=" Start Progressbar ",
command=start_progressbar).grid(column=0, row=1, sticky='W')
ttk.Button(buttons_frame, text=" Stop immediately ",
command=stop_progressbar).grid(column=0, row=2, sticky='W')
ttk.Button(buttons_frame, text=" Stop after second ",
command=progressbar_stop_after).grid(column=0, row=3, sticky='W')
```

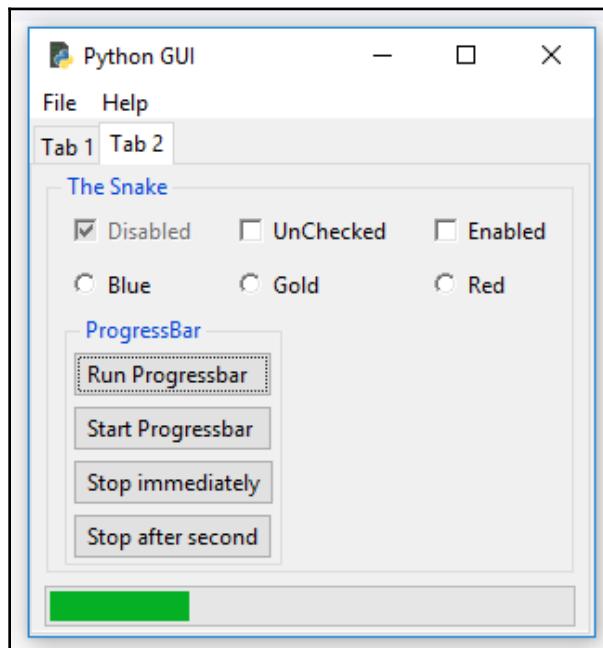
9. Add additional padding for the children of buttons_frame in a loop:

```
for child in buttons_frame.winfo_children():
    child.grid_configure(padx=2, pady=2)
```

10. Add additional padding for all children of Tab2:

```
for child in mighty2.winfo_children():
    child.grid_configure(padx=8, pady=2)
```

11. Run the code. The following GUI is obtained after clicking the **Run Progressbar** button:



Now, let's go behind the scenes to understand the code better.

How it works...

First, we imported `sleep`, otherwise the `Progressbar` would be too fast to be seen. But, be careful when using `sleep` as it can freeze the GUI. We are using it here to simulate a long-running process, which is typically where a `Progressbar` is used.

We then create a `ttk.Progressbar` widget and assign it to `Tab2`.

We create our own callback function, `run_progressbar`, in which we start at 0, loop using `sleep`, and, once we reach the maximum value we have set to 100, and once `Progressbar` has reached the end, we reset it to 0 so `Progressbar` will appear empty again.

We create another function, `start_progressbar`, and in it we use the `ttk.Progressbar` built-in `start` method. If we do not call the `stop` method while `Progressbar` is running, once it has reached the end, it will start to run all over again from the beginning in an endless loop until `stop` has been called.

The `stop_progressbar` function stops `Progressbar` immediately.

The `progressbar_stop_after` function delays the stopping by a certain amount of time. We defaulted it to 1000 milliseconds, which is 1 second, but a different value can be passed into this function.

We achieve this delay by calling the `after` function on the reference to our main GUI window, which we named `win`.

These four functions show us two ways to start and stop `Progressbar`.



Calling the `Stop` functions on the `start_progressbar` function does not stop it, though; it will complete the loop.

We created four new buttons and assigned our functions to their `command` attribute. Clicking the buttons now calls those functions.

We've successfully learned how to create `Progressbar` and start and stop it. Now, let's move on to the next recipe.

How to use the canvas widget

This recipe shows how to add dramatic color effects to our GUI by using the `tkinter` `canvas` widget.

Getting ready

We will improve our previous code from `GUI_tooltip.py`, and we'll improve the look of our GUI by adding some more colors to it.

How to do it...

First, we will create a third tab in our GUI in order to isolate our new code.

Here is the code to create the new third tab:

1. Open `GUI_tooltip.py` and save the module as `GUI_canvas.py`.
2. Create a third tab control:

```
tabControl = ttk.Notebook(win)           # Create Tab Control

tab1 = ttk.Frame(tabControl)             # Create a tab
tabControl.add(tab1, text='Tab 1')       # Add the tab

tab2 = ttk.Frame(tabControl)
tabControl.add(tab2, text='Tab 2')       # Add a second tab

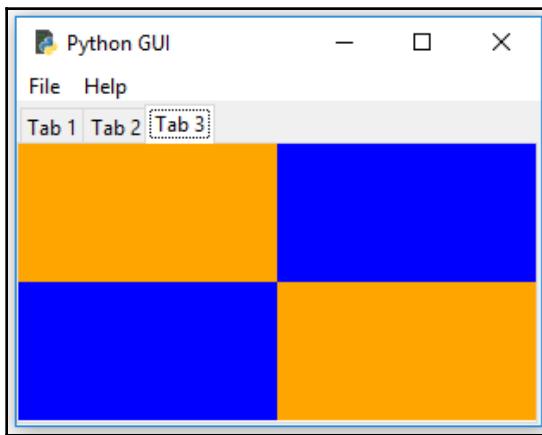
tab3 = ttk.Frame(tabControl)
tabControl.add(tab3, text='Tab 3')       # Add a third tab

tabControl.pack(expand=1, fill="both")   # Pack to make tabs visible
```

3. Next, we use another built-in widget of `tkinter`, called `Canvas`. A lot of people like this widget because it has powerful capabilities:

```
# Tab Control 3 -----
tab3_frame = tk.Frame(tab3, bg='blue')
tab3_frame.pack()
for orange_color in range(2):
    canvas = tk.Canvas(tab3_frame, width=150, height=80,
                       highlightthickness=0, bg='orange')
    canvas.grid(row=orange_color, column=orange_color)
```

- Run the `GUI_canvas.py` file. The following GUI is obtained after running the code:



Now, let's go behind the scenes to understand the code better.

How it works...

After we have created the new tab, we place a regular `tk.Frame` into it and assign it a background color of blue. In the loop, we create two `tk.Canvas` widgets, making their color orange and assigning them to the grid coordinates `0, 0` and `1, 1`. This also makes the blue background color of the `tk.Frame` visible in the two other grid locations.

The preceding screenshot shows the result created by running the preceding code and clicking on the new **Tab 3**. It really is orange and blue when you run the code. In a non-color printed book, this might not be visually obvious, but those colors are true; you can trust me on this.

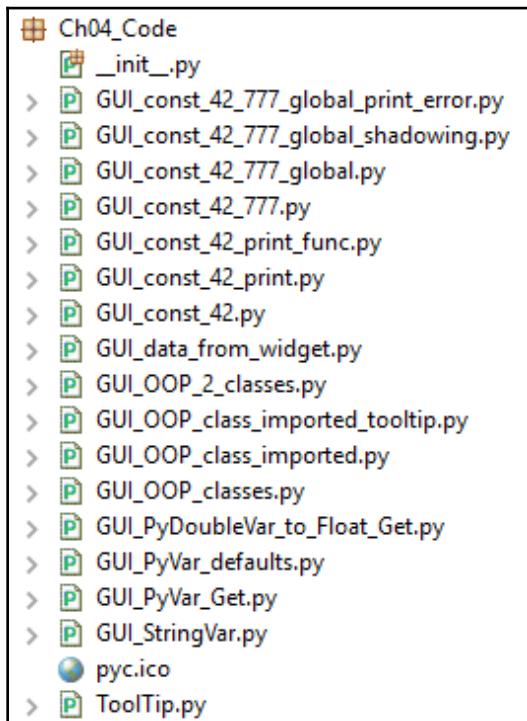
You can check out the graphing and drawing capabilities by searching online. I will not go into the widget in more depth in this book (but it is very cool).

4

Data and Classes

In this chapter, we will save our GUI data into `tkinter` variables. We will also start using **object-oriented programming (OOP)**, writing our own classes in Python. This will lead us to creating reusable OOP components. By the end of this chapter, you will know how to save data from the GUI into local `tkinter` variables. You will also learn how to display tooltips over widgets, which give the user additional information. Knowing how to do this makes our GUI more functional and easier to use.

Here is an overview of the Python modules for this chapter:



In this chapter, we will use data and OOP classes using Python 3.7 and above. We will cover the following recipes:

- How to use `StringVar()`
- How to get data from a widget
- Using module-level global variables
- How coding in classes can improve the GUI
- Writing callback functions
- Creating reusable GUI components

How to use `StringVar()`

There are built-in programming types in `tkinter` that differ slightly from the Python types we are used to programming with. `StringVar()` is one such `tkinter` type. This recipe will show you how to use the `StringVar()` type.

Getting ready

In this recipe, you will learn how to save data from the `tkinter` GUI into variables so we can use that data. We can set and get their values, which is very similar to how you would use the Java getter/setter methods.

Here are some of the types of code in `tkinter`:

<code>strVar = StringVar()</code>	Holds a string; the default value is an empty string ("")
<code>intVar = IntVar()</code>	Holds an integer; the default value is 0
<code>dbVar = DoubleVar()</code>	Holds a float; the default value is 0.0
<code>blVar = BooleanVar()</code>	Holds a Boolean, it returns 0 for False and 1 for True



Different languages call numbers with decimal points `float` or `double`. `tkinter` calls them `DoubleVar`, which is known in Python as the `float` data type. Depending on the level of precision, `float` and `double` data can be different. Here, we are translating `DoubleVar` of `tkinter` into a Python `float` type.

This becomes clearer when we add a `DoubleVar` with a Python `float` and look at the resulting type, which is a Python `float` and no longer a `DoubleVar`.

How to do it...

We will create a `DoubleVar` of `tkinter` variable and add a `float` number literal to it using the `+` operator. After that, we will look at the resulting Python type.

Here are the steps to see the different `tkinter` data types:

1. Create a new Python module and name it
`GUI_PyDoubleVar_to_Float_Get.py`.
2. At the top of the `GUI_PyDoubleVar_to_Float_Get.py` module,
import `tkinter`:

```
import tkinter as tk
```

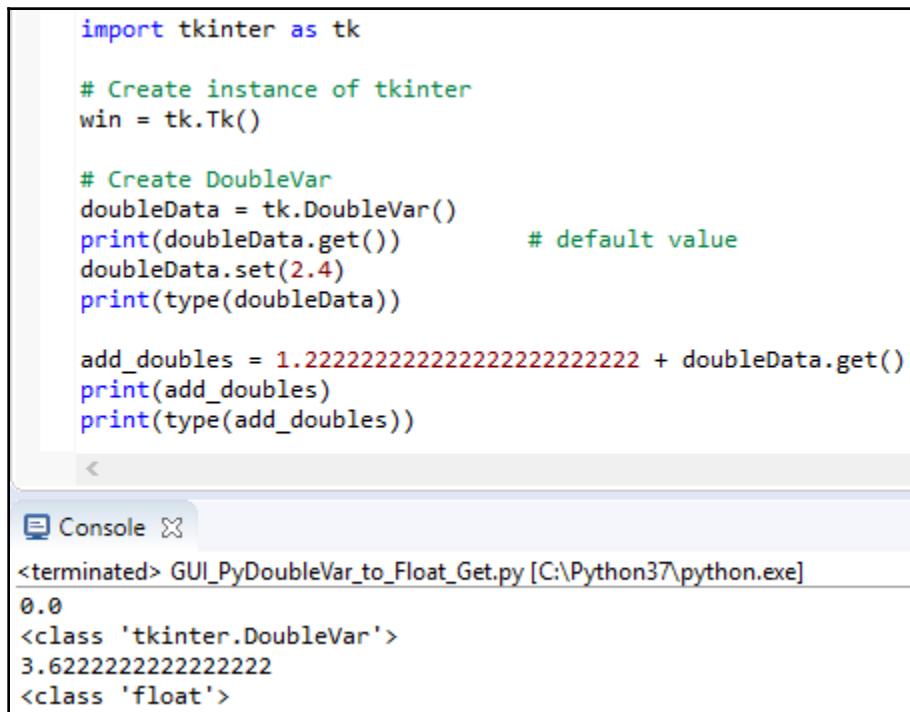
3. Create an instance of the `tkinter` class:

```
win = tk.Tk()
```

4. Create a `DoubleVar` and give it a value:

```
doubleData = tk.DoubleVar()  
print(doubleData.get())  
doubleData.set(2.4)  
print(type(doubleData))  
  
add_doubles = 1.2222222222222222 + doubleData.get()  
print(add_doubles)  
print(type(add_doubles))
```

5. The following screenshot shows the final `GUI_PyDoubleVar_to_Float_Get.py` code and the output after running the code:



The screenshot shows a code editor window with Python code and a terminal window below it. The code imports `tkinter`, creates a `DoubleVar` object, prints its current value (0.0), sets it to 2.4, and then adds it to a floating-point number (1.2222222222222222). The terminal window shows the execution of the script and the resulting output: 0.0, <class 'tkinter.DoubleVar'>, 3.622222222222222, and <class 'float'>.

```
import tkinter as tk

# Create instance of tkinter
win = tk.Tk()

# Create DoubleVar
doubleData = tk.DoubleVar()
print(doubleData.get())          # default value
doubleData.set(2.4)
print(type(doubleData))

add_doubles = 1.2222222222222222 + doubleData.get()
print(add_doubles)
print(type(add_doubles))

<<
<!-->

```

Console

```
<terminated> GUI_PyDoubleVar_to_Float_Get.py [C:\Python37\python.exe]
0.0
<class 'tkinter.DoubleVar'>
3.622222222222222
<class 'float'>
```

We can do the same with `tkinter` with regards to strings.

We will create a new Python module as follows:

1. Create a new Python module and name it `GUI_StringVar.py`.
2. At the top of the `GUI_StringVar.py` module, import `tkinter`:

```
import tkinter as tk
```

3. Create an instance of the `tkinter` class:

```
win = tk.Tk()
```

4. Assign a `StringVar` of `tkinter` to the `strData` variable:

```
strData = tk.StringVar()
```

5. Set the `strData` variable:

```
strData.set('Hello StringVar')
```

6. Get the value of the `strData` variable and save it in `varData`:

```
varData = strData.get()
```

7. Print out the current value of `strData`:

```
print(varData)
```

8. The following screenshot shows the final `GUI_StringVar.py` code and the output after running the code:

The screenshot shows a code editor window with the following Python code:

```
import tkinter as tk

# Create instance of tkinter
win = tk.Tk()

# Assign tkinter Variable to strData variable
strData = tk.StringVar()

# Set strData variable
strData.set('Hello StringVar')

# Get value of strData variable
varData = strData.get()

# Print out current value of strData
print(varData)
```

Below the code editor is a terminal window titled "Console". The terminal shows the command "<terminated> GUI_StringVar.py [C:\Python37\python.exe]" and the output "Hello StringVar".

Next, we will print the default values of, `IntVar`, `DoubleVar`, and `BooleanVar` types of `tkinter`:

1. Open `GUI_StringVar.py` and save the module as `GUI_PyVar_defaults.py`.
2. Add the following lines of code toward the bottom of this module:

```
print(tk.IntVar())
print(tk.DoubleVar())
print(tk.BooleanVar())
```

3. The following screenshot shows the final `GUI_PyVar_defaults.py` code and the output after running the `GUI_PyVar_defaults.py` code file:

The screenshot shows a code editor window with the following content:

```
# Print out current value of strData
print(varData)

# Print out the default tkinter variable values
print(tk.IntVar())
print(tk.DoubleVar())
print(tk.BooleanVar())
```

Below the code editor is a terminal window titled "Console". The terminal output shows the execution of the script and its results:

```
<terminated> GUI_PyVar_defaults.py [C:\Python37\python.exe]
Hello StringVar
PY_VAR1
PY_VAR2
PY_VAR3
```

The steps to print the default `tkinter` variable value are as follows:

1. Create a new Python module and name it `GUI_PyVar_Get.py`.
2. Type the following code into the module:

```
import tkinter as tk
# Create instance of tkinter
win = tk.Tk()
# Print out the default tkinter variable values
intData = tk.IntVar()
print(intData)
print(intData.get())
# Set a breakpoint here to see the values in the debugger
```

```
print()
```

3. Run the code, optionally setting a breakpoint in your IDE in the final `print()` statement:

The screenshot shows a PyDev IDE window with the following components:

- Variables View:** Shows two variables:
 - `intData`: Value `IntVar: PY_VAR0`
 - `intData.get()`: Value `int: 0`
- Breakpoints View:** Shows a single breakpoint at the line `print(intData)`.
- Expressions View:** Shows the value `int: 0`.
- Code Editor:** Displays the Python script `GUI_PyVar_Get.py` with the following code:

```
# Print out the default tkinter variable values
intData = tk.IntVar()
print(intData)
print(intData.get())
```
- Console View:** Shows the output of the script:

```
warning: Debugger speedups using cython not found. Run pydev debugger: starting (pid: 7984)
PY_VAR0
0
```

Let's go behind the scenes to understand the code better.

How it works...

In the Eclipse PyDev console, toward the bottom of the screenshot for `GUI_StringVar.py` in *step 8*, we can see the output printed to the console, which is **Hello StringVar**. This shows us that we have to call the `get()` method to get the data.

As can be seen in the screenshot of `GUI_PyVar_defaults.py` in *step 3*, the default values do not get printed, as we would have expected when we are not calling `get()`.

The online literature mentions default values, but we won't see those values until we call the `get` method on them. Otherwise, we just get a variable name that automatically increments (for example, `PY_VAR3`, as can be seen in the preceding screenshot of `GUI_PyVar_defaults.py`).



Assigning the `tkinter` type to a Python variable does not change the outcome. We still do not get the default value until we call `get()` on this variable.

The value is `PY_VAR0`, not the expected `0`, until we call the `get` method. Now we can see the default value. We did not call `set`, so we see the default value automatically assigned to each `tkinter` type once we call the `get` method on each type.

Note how the default value of `0` gets printed to the console for the `IntVar` instance that we saved in the `intData` variable. We can also see the values in the Eclipse PyDev debugger window at the top of the screenshot.

First, we import the `tkinter` module and alias it to the name `tk`. Next, we use this alias to create an instance of the `Tk` class by appending parentheses to `Tk`, which calls the constructor of the class. This is the same mechanism as calling a function; only here, we create an instance of a class.

Usually, we use this instance assigned to the `win` variable to start the main event loop later in the code, but here, we are not displaying a GUI; rather, we are demonstrating how to use the `StringVar` type of `tkinter`.



We still have to create an instance of `Tk()`. If we comment out this line, we will get an error from `tkinter`, so this call is necessary.

Then, we create an instance of the `StringVar` type `tkinter` and assign it to our Python `strData` variable. After that, we use our variable to call the `set()` method on `StringVar` and after setting it to a value, we get the value, save it in a new variable named `varData`, and then print out its value. We've successfully learned how to use `StringVar()`. Now let's move on to the next recipe.

How to get data from a widget

When the user enters data, we want to do something with it in our code. This recipe shows how to capture data in a variable. In the previous recipe, we created several `tkinter` class variables. They were standalone. Now, we are connecting them to our GUI, using the data we get from the GUI, and storing them in Python variables.

Getting ready

We will continue using the Python GUI we were building in Chapter 3, *Look and Feel Customization*. We'll reuse and enhance the code from `GUI_progressbar.py` from that chapter.

How to do it...

We will assign a value from our GUI to a Python variable:

1. Open `GUI_progressbar.py` from Chapter 3, *Look and Feel Customization*, and save the module as `GUI_data_from_widget.py`.
2. Add the following code toward the bottom of our module. Just above the main event loop, add `strData`:

```
strData = spin.get()  
print("Spinbox value: " + strData)
```

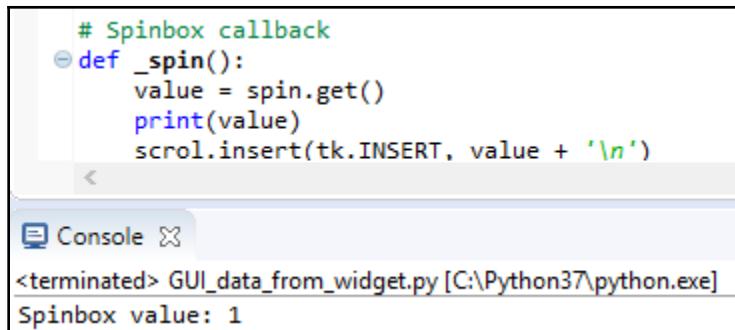
3. Add code to place the cursor into the name entry:

```
name_entered.focus()
```

4. Start the GUI:

```
win.mainloop()
```

5. Running the code gives us the following result:



The screenshot shows a code editor with Python code and a terminal window. The code defines a spinbox callback function named `_spin` that prints the current value of the spinbox and inserts it into a text area. The terminal window shows the output: <terminated> GUI_data_from_widget.py [C:\Python37\python.exe] Spinbox value: 1.

```
# Spinbox callback
def _spin():
    value = spin.get()
    print(value)
    scrol.insert(tk.INSERT, value + '\n')
```

```
Console X
<terminated> GUI_data_from_widget.py [C:\Python37\python.exe]
Spinbox value: 1
```



We placed our code above the GUI main event loop, so the printing happens before the GUI becomes visible. We would have to place the code into a callback function if we wanted to print out the current value after displaying the GUI and changing the value of the Spinbox control.

We will retrieve the current value of the Spinbox control:

1. We create our Spinbox widget using the following code, hard-coding the available values into it:

```
# Adding a Spinbox widget using a set of values
spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5, bd=8,
command=_spin)
spin.grid(column=0, row=2)
```

2. We can also move the hard-coding of the data out of the creation of the Spinbox class instance and set it later:

```
# Adding a Spinbox widget assigning values after creation
spin = Spinbox(mighty, width=5, bd=8, command=_spin)
spin['values'] = (1, 2, 4, 42, 100)
spin.grid(column=0, row=2)
```

It does not matter how we create our widget and insert data into it because we can access this data by using the `get()` method on the instance of the widget.

Let's go behind the scenes to understand the code better.

How it works...

In order to get the values out of our GUI written using `tkinter`, we use the `get()` method of `tkinter` on an instance of the widget we wish to get the value from.

In the preceding example, we used the `Spinbox` control, but the principle is the same for all widgets that have a `get()` method.

Once we have got the data, we are in a pure Python world, and `tkinter` did serve us well in building our GUI. Now that we know how to get the data out of our GUI, we can use this data.

We've successfully learned how to get data from a widget. Now let's move on to the next recipe.

Using module-level global variables

Encapsulation is a major strength in any programming language, enabling us to program using OOP. Python is both OOP-friendly as well as procedural. We can create `global` variables that are localized to the module they reside in. They are `global` only to this module, which is one form of encapsulation. Why do we want this? Because as we add more and more functionality to our GUI, we want to avoid naming conflicts that could result in bugs in our code.



We do not want naming clashes creating bugs in our code! Namespaces are one way to avoid these bugs, and in Python, we can do this by using Python modules (which are unofficial namespaces).

Getting ready

We can declare module-level **globals** in any module just above and outside functions.

We then have to use the `global` Python keyword to refer to them. If we forget to use `global` in functions, we will accidentally create new local variables. This would be a bug and something we really do not want to do.



Python is a dynamic, strongly typed language. We will notice bugs such as this (forgetting to scope variables with the `global` keyword) only at runtime.

How to do it...

Add the following code to the GUI we used in the previous recipe, *How to get data from a widget*, creating a module-level global variable. We use the all-uppercase convention for constants:



You can find more information in **PEP 8 -- Style Guide for Python Code** at <https://www.python.org/dev/peps/pep-0008/#constants>.

1. Open `GUI_data_from_widget.py` and save the module as `GUI_const_42_print.py`.
2. Add the constant variable at the top and the `print` statement at the bottom of the module:

```
GLOBAL_CONST = 42
# ...
print(GLOBAL_CONST)
```

3. Running the code results in a printout of the `global`. Note **42** being printed to the Eclipse console (`GUI_const_42_print.py`):

A screenshot of the Eclipse IDE interface. The code editor shows a Python script with the following content:

```
# Printing the Global works
print(GLOBAL_CONST)

name_entered.focus()
#=====
# Start GUI
#=====
win.mainloop()
```

The terminal window below it is titled "Console" and shows the output of the script:

```
<terminated> GUI_const_42_print.py [C:\Python37\python.exe]
42
```

Add the `usingGlobal` function toward the bottom of the module:

1. Open `GUI_const_42_print.py` and save the module as `GUI_const_42_print_func.py`.
2. Add the function and then call it:

```
def usingGlobal():
    print(GLOBAL_CONST)
# call the function
usingGlobal()
```

3. The following screenshot shows the final `GUI_const_42_print_func.py` code and the output after running the code:

The screenshot shows a code editor window with the following content:

```
def usingGlobal():
    print(GLOBAL_CONST)

# call function
usingGlobal()
<
```

Below the code editor is a terminal window titled "Console" with the following output:

```
<terminated> GUI_const_42_print_func.py [C:\Python37\python.exe]
42
```

In the preceding code snippet, we use the module-level `global`. It is easy to make a mistake by *shadowing* the `global`, as demonstrated in the following code:

1. Open `GUI_const_42_print_func.py` and save the module as `GUI_const_42_777.py`.
2. Add the declaration of the constant within the function:

```
def usingGlobal():
    GLOBAL_CONST = 777
    print(GLOBAL_CONST)
```

3. The following screenshot shows the final `GUI_const_42_777.py` code and the output after running the code:

The screenshot shows a code editor window with the following content:

```
def usingGlobal():
    GLOBAL_CONST = 777
    print(GLOBAL CONST)
<
```

Below the code editor is a terminal window titled "Console" with the following output:

```
<terminated> GUI_const_42_777.py [C:\Python37\python.exe]
777
```

Note how **42** becomes **777**, even though we are using the same variable name.



There is no compiler in Python that warns us if we override `global` variables in a local function. This can lead to difficulties in debugging at runtime.

If we try to print out the value of the global variable, without using the `global` keyword, we get an error:

1. Open `GUI_const_42_777.py` and save the module as `GUI_const_42_777_global_print_error.py`.
2. Comment out `global` and try to print:

```
def usingGlobal():
    # global GLOBAL_CONST
    print(GLOBAL_CONST)
    GLOBAL_CONST = 777
    print(GLOBAL_CONST)
```

3. Run the code and observe the output:

A screenshot of a terminal window titled "Console". The window shows a Python traceback for a script named "GUI_const_42_777_global_print_error.py". The traceback indicates that the error occurred in the "usingGlobal()" function, which was called from another part of the code. The specific error message is "UnboundLocalError: local variable 'GLOBAL_CONST' referenced before assignment".

```
Console ✘
<terminated> GUI_const_42_777_global_print_error.py [C:\Python37\python.exe]
Traceback (most recent call last):
  File "C:\Eclipse Oxygen workspace Packt 3rd GUI BOOK\3rd Edition Python GUI
    usingGlobal()
  File "C:\Eclipse Oxygen workspace Packt 3rd GUI BOOK\3rd Edition Python GUI
    print(GLOBAL_CONST)
UnboundLocalError: local variable 'GLOBAL_CONST' referenced before assignment
```

When we qualify our local variable with the `global` keyword, we can print out the value of the global variable and overwrite this value locally:

1. Open `GUI_const_42_777_global.py`.
2. Add the following code:

```
def usingGlobal():
    global GLOBAL_CONST
    print(GLOBAL_CONST)
    GLOBAL_CONST = 777
    print(GLOBAL_CONST)
```

3. Run the code and observe the output:

The screenshot shows a code editor with a script named `GUI_const_42_777_global.py`. The code defines a function `usingGlobal` that prints the current value of `GLOBAL_CONST`, changes its value to 777, and then prints it again. Below the code, a command-line interface labeled "Console" shows the execution of the script. The output consists of two lines: "42" and "777".

```
def usingGlobal():
    global GLOBAL_CONST
    print(GLOBAL_CONST)
    GLOBAL_CONST = 777
    print(GLOBAL_CONST)

# call function
usingGlobal()

<terminated> GUI_const_42_777_global.py [C:\Python37\python.exe]
42
777
```

We might believe that the value of the global variable is local to our function.

1. Open `GUI_const_42_777_global.py` and save as `GUI_const_42_777_global_shadowing.py`.
2. Add `print('GLOBAL_CONST:', GLOBAL_CONST)` below the function.

3. Run the code and observe the output:

The screenshot shows a terminal window titled "Console". The code in the terminal is:

```
# call function
usingGlobal()

# call the global from outside the function
print('GLOBAL_CONST:', GLOBAL_CONST)
```

The output of the code is:

```
<terminated> GUI_const_42_777_global_shadowing.py [C:\Python37\python.exe]
42
777
GLOBAL_CONST: 777
```

Let's go behind the scenes to understand the code better.

How it works...

We define a `global` variable at the top of our module, and we print out its value later, toward the bottom of our module.

That works. We then define a function and print out the value of the `global` within the function by using the `global` keyword. If we forget to use the `global` keyword, we are creating a new, local variable. When we change the value of the `global` inside the function, this actually changes the `global` variable. As we can see, even outside of our function the `global` value has changed.

`global` variables can be very useful when programming small applications. They can help us make data available across methods and functions within the same Python module and, sometimes, the overhead of OOP is not justified.

As our programs grow in complexity, the benefit we gain from using `globals` can quickly diminish.



It is best to avoid `globals` and accidentally shadowing variables by using the same name in different scopes. We can use OOP instead of using `global` variables.

We have played around with `global` variables within procedural code and have learned how it can lead to hard-to-debug bugs. In the next recipe, we will move on to OOP, which can eliminate such bugs.

How coding in classes can improve the GUI

So far, we have been coding in a procedural style. This is a quick scripting method we can do in Python. When our code gets larger and larger, we need to advance to coding in OOP.

Why?

Because, among many other benefits, OOP allows us to move code around by using methods. Once we use classes, we no longer have to physically place the code above the code that calls it. This gives us great flexibility in organizing our code. We can write the related code next to the other code and no longer have to worry that the code will not run because the code does not sit above the code that calls it. We can take that to some rather fancy extremes by coding up modules that refer to methods that are not being created within that module. They rely on the runtime state having created those methods during the time the code runs.



If the methods we call have not been created by that time, we get a runtime error.

Getting ready

We will turn our entire procedural code into OOP very simply. We just turn it into a class, indent all the existing code, and prepend `self` to all variables.

It is very easy.

While at first it might feel a little bit annoying having to prepend everything with the `self` keyword, making our code more verbose (*hey, we are wasting so much paper...*), in the end it is worth it.

How to do it...

Note that in the Eclipse IDE, the PyDev editor hints at coding problems by highlighting them in red on the right-hand side portion of the code editor.

1. Open `GUI_const_42_777_global.py` and save the module as `GUI_OOP_classes.py`.
2. Highlight the entire code below the imports and indent it by four spaces.

3. Add `class OOP()` : above the indented code.
4. Look at all of the red errors in the code editor on the right-hand side:

The screenshot shows the Eclipse Oxygen IDE interface. The top window is titled "GUI_OOP_classes" and contains Python code for creating a GUI with tabs and a button. Several lines of code are highlighted in red, indicating syntax errors. The bottom window is titled "Console" and shows the command-line output of running the script, which includes a stack trace and a "win.mainloop()" call.

```

@class OOP():
    # Create instance
    win = tk.Tk()

    # Add a title
    win.title("Python GUI")

   .tabControl = ttk.Notebook(win)      # Create Tab Control

    tab1 = ttk.Frame(tabControl)        # Create a tab
    tabControl.add(tab1, text="Tab 1")   # Add the tab
    tab2 = ttk.Frame(tabControl)        # Add a second tab
    tabControl.add(tab2, text="Tab 2")   # Make second tab visible

    tabControl.pack(expand=1, fill="both") # Pack to make visible

    # LabelFrame using tab1 as the parent
    mighty = ttk.LabelFrame(tab1, text=' Mighty Python ')
    mighty.grid(column=0, row=0, padx=8, pady=4)

    # Modify adding a Label using mighty as the parent instead of win
    a_label = ttk.Label(mighty, text="Enter a name:")
    a_label.grid(column=0, row=0, sticky='W')

    # Modified Button Click Function
    def click_me():
        action.configure(text='Hello ' + name.get() + ' ' +
                        number_chosen.get())

```

```

777
Traceback (most recent call last):
File "C:\Eclipse Oxygen workspace Packt 3rd GUI BOOK\3rd.Edition Python GUI Programming Cookbook\Ch04 Code\GUI OOP classes.py", line 229, in <module>
    win.mainloop()

```

We have to prepend all the variables with the `self` keyword and also bind the functions to the class by using `self`, which officially and technically turns the functions into methods.

Let's prefix everything with `self` to fix all of the red so we can run our code again:

1. Open `GUI_OOP_classes.py` and save the module as `GUI_OOP_2_classes.py`.
2. Add the `self` keyword wherever it is needed, for example, `click_me(self)`.
3. Run the code and observe it:

```

# Modified Button Click Function
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get() + ' ' +
                        self.number_chosen.get())

```

Once we do this for all of the errors highlighted in red, we can run our Python code again. The `click_me` function is now bound to the class and has officially become a method. We are no longer getting any errors that prevent the code from running.

Now let's add our `ToolTip` class from Chapter 3, *Look and Feel Customization*, into this Python module:

1. Open `GUI_OOP_2_classes.py`.
2. Add the `ToolTip` class from `GUI_tooltip.py` to the top of the following module's import statements:

```
class ToolTip(object):
    def __init__(self, widget, tip_text=None):
        self.widget = widget
    ...
class OOP():
    def __init__(self):
        self.win = tk.Tk()
        ToolTip(self.win, 'Hello GUI')
        # <-- use the ToolTip class here
    ...
    ...
```

Let's go behind the scenes to understand the code better.

How it works...

We are translating our procedural code into object-oriented code. First, we indented the entire code and defined the code to be part of a class, which we named `OOP`. In order to make this work, we have to use the `self` keyword for both variables and methods. Here is a brief comparison of our previous code with the new OOP code using a class:

```
#####
# Our procedural code looked like this:
#####
# Button Click Function
def click_me():
    action.configure(text='Hello ' + name.get() + ' ' +
                    number_chosen.get())

# Adding a Textbox Entry widget
name = tk.StringVar()
name_entered = ttk.Entry(mighty, width=12, textvariable=name)
name_entered.grid(column=0, row=1, sticky='W')

# Adding a Button
action = ttk.Button(mighty, text="Click Me!", command=click_me)
action.grid(column=2, row=1)

ttk.Label(mighty, text="Choose a number:").grid(column=1, row=0)
```

```
number = tk.StringVar()
number_chosen = ttk.Combobox(mighty, width=12,
textvariable=number, state='readonly')
number_chosen['values'] = (1, 2, 4, 42, 100)
number_chosen.grid(column=1, row=1)
number_chosen.current(0)
# ...

*****
The new OOP code looks like this:
*****
class OOP():
    def __init__(self):                      # Initializer method
        # Create instance
        self.win = tk.Tk()                   # notice the self keyword
        ToolTip(self.win, 'Hello GUI')
        # Add a title
        self.win.title("Python GUI")
        self.create_widgets()

    # Button callback
    def click_me(self):
        self.action.configure(text='Hello ' + self.name.get() + ' '
+ self.number_chosen.get())
        # ... more callback methods

    def create_widgets(self):
        # Create Tab Control
        tabControl = ttk.Notebook(self.win)
        tab1 = ttk.Frame(tabControl)           # Create a tab
        tabControl.add(tab1, text='Tab 1')      # Add the tab
        tab2 = ttk.Frame(tabControl)           # Create second tab
        tabControl.add(tab2, text='Tab 2')      # Add second tab
        # Pack to make visible
        tabControl.pack(expand=1, fill="both")

        # Adding a Textbox Entry widget - using self
        self.name = tk.StringVar()
        name_entered = ttk.Entry(mighty, width=12,
textvariable=self.name)
        name_entered.grid(column=0, row=1, sticky='W')
        # Adding a Button - using self
        self.action = ttk.Button(mighty, text="Click Me!",
command=self.click_me)
        self.action.grid(column=2, row=1)
        #

=====
# Start GUI
```

```
#=====
oop = OOP()      # create an instance of the class
                  # use instance variable to call mainloop via oop.win
oop.win.mainloop()
```

We moved the callback methods to the top of the module, inside the new `OOP` class. We moved all the widget-creation code into one rather long method, `create_widgets`, which we call in the initializer of the class. Technically, deep underneath the hood of the low-level code Python does have a constructor, yet Python frees us from any worries about this. It is taken care of for us. Instead, in addition to a real constructor, Python provides us with an initializer, `__init__(self)`. We are strongly encouraged to use this initializer. We can use it to pass in arguments to our class, initializing variables we wish to use inside our class instance.

In the end, we added the `ToolTip` class to the top of our module just below the `import` statements.



In Python, several classes can exist within the same Python module and the module name does not have to be the same as the class name.

Here, in this recipe, we can see that more than one class can live in the same Python module.

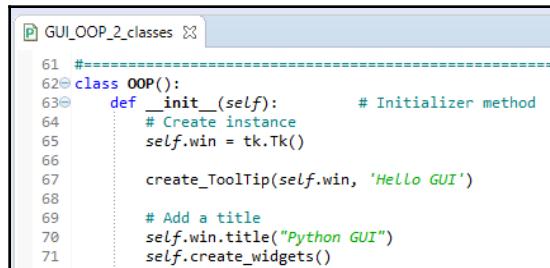
Cool stuff, indeed! Here are two screenshots of the two classes residing in the same module:

A screenshot of a code editor window titled "GUI_OOP_2_classes". The code is as follows:

```
20  =====
21@ class ToolTip(object):
22@     def __init__(self, widget):
23         self.widget = widget
24         self.tip_window = None
25
26@     def show_tip(self, tip_text):
27         "Display text in a tooltip window"
```

The code editor shows line numbers 20 through 27. Lines 20 and 21 define the `ToolTip` class. Lines 22 and 23 implement the `__init__` method, setting `self.widget` and `self.tip_window`. Lines 26 and 27 implement the `show_tip` method, with a docstring indicating it displays text in a tooltip window.

Both the `ToolTip` class and the `OOP` class reside within the same Python module, `GUI_OOP_2_classes.py`:



```
61 =====
62 class OOP():
63     def __init__(self):          # Initializer method
64         self.win = tk.Tk()
65
66         create_ToolTip(self.win, 'Hello GUI')
67
68         # Add a title
69         self.win.title("Python GUI")
70
71         self.create_widgets()
```

In this recipe, we advanced our procedural code into OOP code. Python enables us to write code in both a practical and a procedural style, like the C programming language style. At the same time, we have the option to code in an OOP style, like the Java, C#, and C++ style.

We've successfully learned how coding in classes can improve the GUI. Now let's move on to the next recipe.

Writing callback functions

At first, callback functions can seem to be a little bit intimidating. You call the function, passing it some arguments, and then the function tells you that it is really very busy and it will call you back!

You wonder: will this function ever call me back? And how long do I have to wait? In Python, even callback functions are easy and, yes, they usually do call you back. They just have to complete their assigned task first (*hey, it was you who coded them in the first place...*).

Let's learn a little bit more about what happens when we code callbacks into our GUI. Our GUI is event-driven. After it has been created and displayed onscreen, it typically sits there waiting for an event to happen. It is basically waiting for an event to be sent to it. We can send an event to our GUI by clicking one of its buttons. This creates an event and, in a sense, we called our GUI by sending it a message.

Now, what is supposed to happen after we send a message to our GUI? What happens after clicking the button depends on whether we created an event handler and associated it with this button. If we did not create an event handler, clicking the button will have no effect. The event handler is a callback function (or method, if we use classes). The callback method is also sitting there passively, like our GUI, waiting to be invoked. Once our GUI's button is clicked, it will invoke the callback.

The callback often does some processing and, when done, it returns the result to our GUI.



In a sense, we can see that our callback function is calling our GUI back.

Getting ready

The Python interpreter runs through all the code in a module once, finding any syntax errors and pointing them out. You cannot run your Python code if you do not have the syntax right. This includes indentation (if not resulting in a syntax error, incorrect indentation usually results in a bug).

On the next parsing round, the interpreter interprets our code and runs it.

At runtime, many GUI events can be generated, and it is usually callback functions that add functionality to GUI widgets.

How to do it...

Here is the callback for the Spinbox widget:

1. Open `GUI_OOP_2_classes.py`.
 2. Observe the `spin(self)` method in the code:

```
37     # Spinbox callback
38     def _spin(self):
39         value = self.spin.get()
40         print(value)
41         self.scrol.insert(tk.INSERT, value + '\n')
42
43     # Adding a Spinbox widget
44     self.spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5, bd=9, command=self._spin)
45     self.spin.grid(column=0, row=2)
```

Let's go behind the scenes to understand the code better.

How it works...

We create a callback method in the `OOP` class that gets called when we select a value from the `Spinbox` widget because we bind the method to the widget via the `command` argument (`command=self._spin`). We use a leading underscore to hint at the fact that this method is meant to be respected like a private Java method.

Python intentionally avoids language restrictions, such as private, public, friend, and so on. In Python, we use naming conventions instead. Leading and trailing double underscores surrounding a keyword are expected to be restricted to the Python language, and we are expected not to use them in our own Python code.

However, we can use a leading underscore prefix with a variable name or function to provide a hint that this name is meant to be respected as a private helper.

At the same time, we can postfix a single underscore if we wish to use what otherwise would be built-in Python names. For example, if we wished to abbreviate the length of a list, we could do the following:

```
len_ = len(aList)
```



Often, the underscore is hard to read and easy to overlook, so this might not be the best idea in practice.

We've successfully learned how to write callback functions. Now let's move on to the next recipe.

Creating reusable GUI components

We will create reusable GUI components using Python. In this recipe, we will keep it simple by moving our `ToolTip` class into its own module. Then, we will import and use it to display tooltips over several widgets of our GUI.

Getting ready

We are building our code from *Chapter 3, Look and Feel*

Customization: GUI_tooltip.py. We will start by breaking out our `ToolTip` class into a separate Python module.

How to do it...

We will create a new Python module and place the ToolTip class code into it and then import this module into our primary module:

1. Open `GUI_OOP_2_classes.py` and save the module as `GUI_OOP_class_imported_tooltip.py`.
2. Break out the ToolTip code from `GUI_tooltip.py` into a new Python module and name the module `ToolTip.py`.
3. Import the `ToolTip` class into `GUI_OOP_class_imported_tooltip.py`:

```
from Ch04_Code.ToolTip import ToolTip
```

4. Add the following code to `GUI_OOP_class_imported_tooltip.py`:

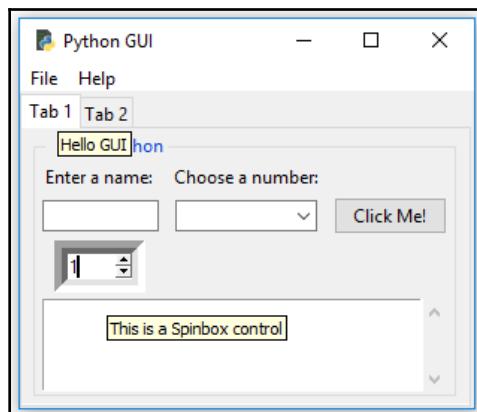
```
ToolTip(self.win, 'Hello GUI')

# Add a ToolTip to the Spinbox
ToolTip(self.spin, 'This is a Spinbox control')

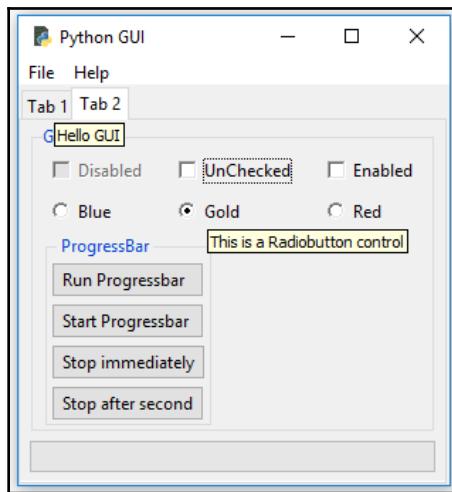
# Add tooltips to more widgets
ToolTip(self.name_entered, 'This is an Entry control')
ToolTip(self.action, 'This is a Button control')
ToolTip(self.scrol, 'This is a ScrolledText control')

# Tab 2
ToolTip(curRad, 'This is a Radiobutton control')
```

5. Run the code and hover the mouse over the different widgets:



This also works on the second tab:



Let's go behind the scenes to understand the code better.

How it works...

First, we created a new Python module and placed the `ToolTip` class into this new module. Then, we imported this `ToolTip` class into a different Python module. After that, we created several tooltips using the class.

In the preceding screenshots, we can see several `ToolTip` messages being displayed. The one for the main window might appear a little bit annoying, so it is better not to display a `ToolTip` for the main window because we really wish to highlight the functionality of the individual widgets. The main window form has a title that explains its purpose; no need for a `ToolTip`.

Refactoring our common `ToolTip` class code out into its own module helps us reuse this code from other modules. Instead of copy/paste/modify, we use the **DRY principle** and our common code is located in only one place, so when we modify the code, all modules that import it will automatically get the latest version of our module.



DRY is short for **Don't Repeat Yourself**, and we will look at it again in a later chapter. We can do similar things by turning our **Tab 3** image into a reusable component. To keep this recipe's code simple, we removed **Tab 3**, but you can experiment with the code from the previous chapter.

5

Matplotlib Charts

In this chapter, we will create beautiful charts that visually represent data. Depending on the format of the data source, we can plot one or more columns of data in the same chart.

We will be using the Python `Matplotlib` module to create our charts.

In a company I worked for, we had an existing program that collected data for analysis. It was a manual process to load the data into Excel and then generate charts within Excel.

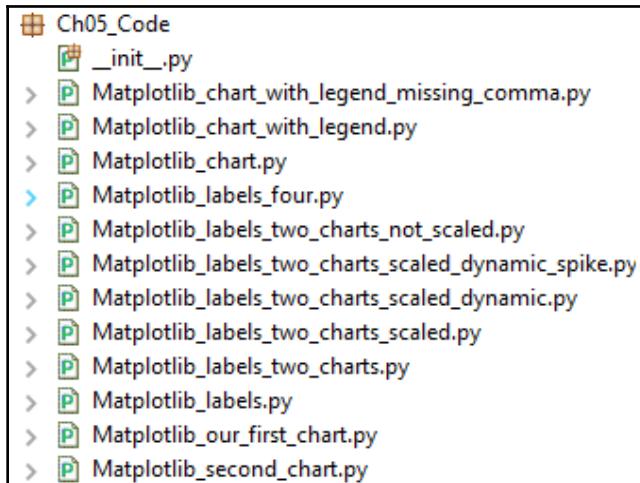
I automated the entire process using Python and `Matplotlib`. With only one click of the mouse, the data got backed up to a network drive and, with another click, the charts got automatically created.

In order to create these graphical charts, we need to download additional Python modules, and there are several ways to install them.

This chapter will explain how to download the `Matplotlib` Python module along with all the other requisite Python modules and the ways in which to do this. After we install the required modules, we will then create our own Pythonic charts.

Visually representing data makes our GUI very useful and great looking and greatly enhances your coding skills. It is also very useful for your management team to represent data visually.

Here is an overview of the Python modules for this chapter:



In this chapter, we will create beautiful charts using Python 3.7 and above with the `Matplotlib` module.



The following URL, <http://matplotlib.org/users/screenshots.html>, is a great place to begin exploring the world of `Matplotlib`, and it teaches us how to create many charts that are not presented in this chapter.

We will cover the following recipes:

- Installing `Matplotlib` using `pip` with the `.whl` extension
- Creating our first chart
- Placing labels on charts
- How to give the chart a legend
- Scaling charts
- Adjusting the scale of charts dynamically

Installing Matplotlib using pip with the .whl extension

The usual way to download additional Python modules is by using `pip`. The `pip` module comes pre-installed with the latest version of Python (3.7 and above).



If you are using an older version of Python, you may have to download both `pip` and `setuptools` by yourself.

This recipe will show how to successfully install `Matplotlib` using `pip`. We will be using the `.whl` extension for this installation, so this recipe will also show you how to install the `wheel` module.

Getting ready

First, let's find out whether you have the `wheel` module already installed. The `wheel` module is necessary to download and install Python packages that have the `.whl` extension.

We can find out what modules we have currently installed using `pip`.

From the Windows Command Prompt, run the `pip list` command:

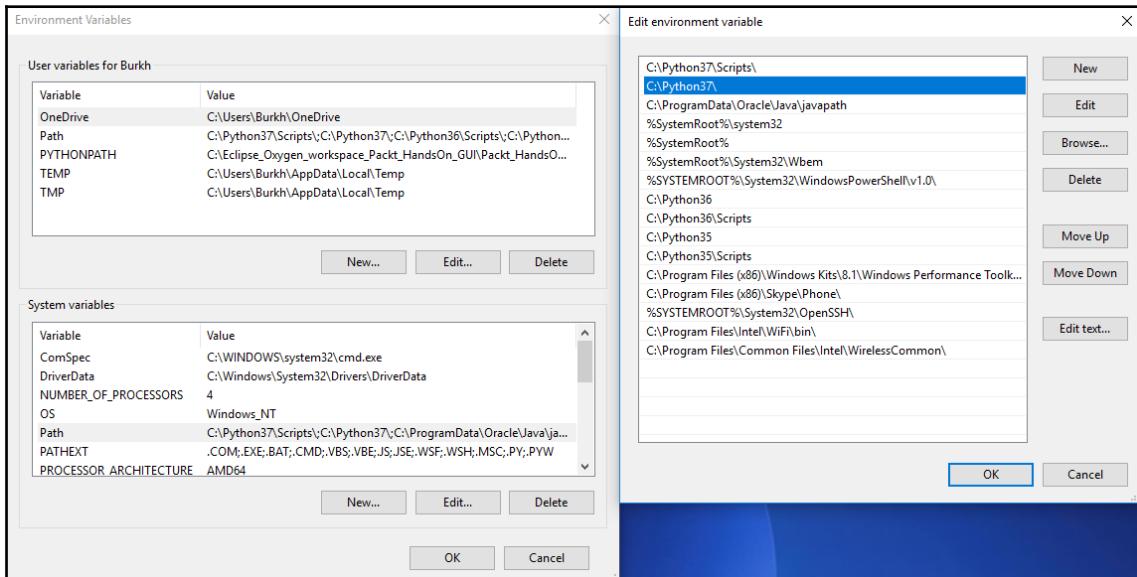
A screenshot of a Windows Command Prompt window titled "Administrator: Command Prompt". The window shows the following text:

```
C:\> Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>pip list
DEPRECATION: The default format will switch to columns
  format=(legacy|columns) in your pip.conf under the [li
pip (9.0.1)
setuptools (28.8.0)

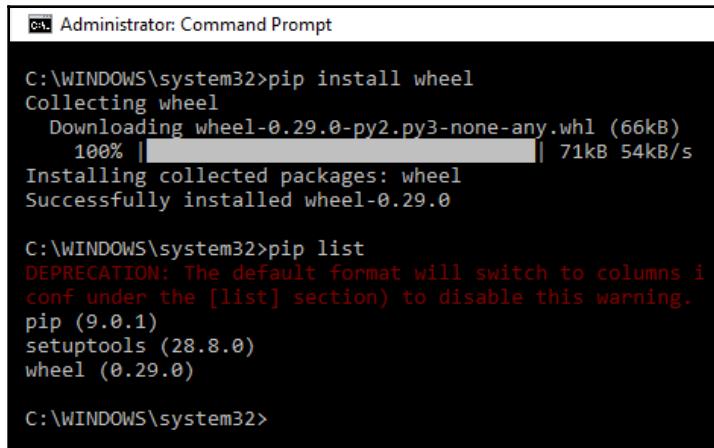
C:\WINDOWS\system32>
```

If you get an error running this command, you might want to check whether Python is on your environmental path. If it currently isn't, add it to **System variables | Path** (bottom-left) by clicking the **Edit...** button. Then, click the **New** button (top-right) and type in the path to your Python installation. Also, add the `C:\Python37\Scripts` directory, as the `pip.exe` file is located there:



If you have more than one version of Python installed, it is a good idea to move Python 3.7 to the top of the list. When we type `pip install <module>`, the first version found in **System variables | Path** might be used and you might get some unexpected errors if an older version of Python is located above Python 3.7.

Let's run `pip install wheel` and then verify whether it has been installed successfully using `pip list`:

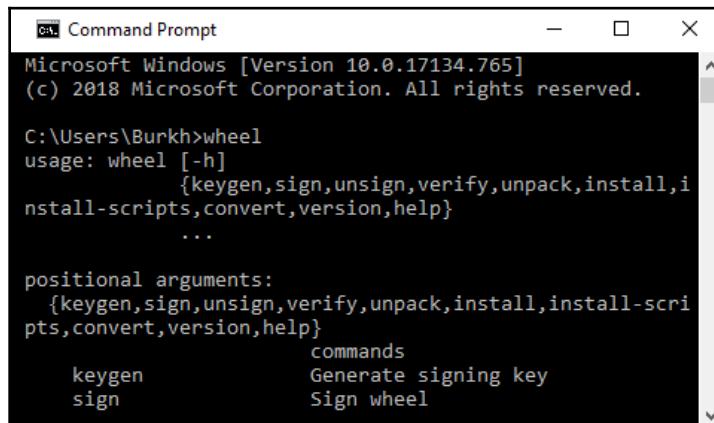


```
C:\WINDOWS\system32>pip install wheel
Collecting wheel
  Downloading wheel-0.29.0-py2.py3-none-any.whl (66kB)
    100% |████████████████████████████████| 71kB 54kB/s
Installing collected packages: wheel
Successfully installed wheel-0.29.0

C:\WINDOWS\system32>pip list
DEPRECATION: The default format will switch to columns in
conf under the [list] section) to disable this warning.
pip (9.0.1)
setuptools (28.8.0)
wheel (0.29.0)

C:\WINDOWS\system32>
```

If running `pip list` does not show `wheel`, try to simply type `wheel` at Command Prompt. This assumes that you have set up your Python path correctly:



```
Microsoft Windows [Version 10.0.17134.765]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Burkh>wheel
usage: wheel [-h]
              {keygen,sign,unsign,verify,unpack,install,i
nstall-scripts,convert,version,help}
              ...
positional arguments:
  {keygen,sign,unsign,verify,unpack,install,install-scri
pts,convert,version,help}
          commands
  keygen           Generate signing key
  sign             Sign wheel
```

If you are really very used to Python 2.7 and insist on running the code in Python 2.7, you can try this trick. After everything is working with Python 3.7, you can rename the 3.7 `python.exe` to `python3.exe` and then have fun using both 2.7 and 3.7 by typing `python.exe` or `python3.exe` in a command window to run the different Python executables. It is a hack. If you really wish to go on this road, you are on your own, but it can work.



How to do it...

With the wheel module installed, we can now proceed with downloading and installing Matplotlib from <http://www.lfd.uci.edu/~gohlke/pythonlibs/>.

1. Download the matching Matplotlib wheel to your hard drive:

Matplotlib, a 2D plotting library.

Requires [numpy](#), [dateutil](#), [pytz](#), [pyparsing](#), [kiwisolver](#), [imagemagick](#).

[matplotlib-2.2.4-cp27-cp27m-win32.whl](#)

[matplotlib-2.2.4-cp27-cp27m-win_amd64.whl](#)

[matplotlib-2.2.4-cp35-cp35m-win32.whl](#)

[matplotlib-2.2.4-cp35-cp35m-win_amd64.whl](#)

[matplotlib-2.2.4-cp36-cp36m-win32.whl](#)

[matplotlib-2.2.4-cp36-cp36m-win_amd64.whl](#)

[matplotlib-2.2.4-cp37-cp37m-win32.whl](#)

[matplotlib-2.2.4-cp37-cp37m-win_amd64.whl](#)

[matplotlib-2.2.4-pp271-pypy_41-win32.whl](#)

[matplotlib-2.2.4-pp370-pp370-win32.whl](#)

[matplotlib-2.2.4.chm](#)

[matplotlib-3.0.3-cp35-cp35m-win32.whl](#)

[matplotlib-3.0.3-cp35-cp35m-win_amd64.whl](#)

[matplotlib-3.0.3.chm](#)

[matplotlib-3.1.0-cp36-cp36m-win32.whl](#)

[matplotlib-3.1.0-cp36-cp36m-win_amd64.whl](#)

[matplotlib-3.1.0-cp37-cp37m-win32.whl](#)

[matplotlib-3.1.0-cp37-cp37m-win_amd64.whl](#)

2. Open Command Prompt and run `pip install <matplotlib wheel>` as shown:

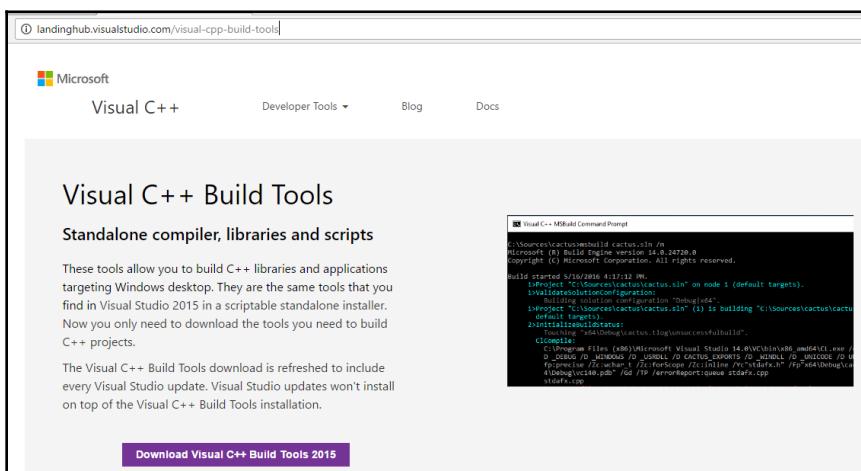
```
C:\Users\Burkhard\Desktop\2nd EDITION PACKT PYTHON GUI COOKBOOK\SW DOWNLOADS>pip install matplotlib-1.5.3-cp36-cp36m-win_amd64.whl
Processing c:\users\burkhard\desktop\2nd edition packt python gui cookbook\sw downloads\matplotlib-1.5.3-cp36-cp36m-win_amd64.whl
Collecting cycler (from matplotlib==1.5.3)
  Downloading cycler-0.10.0-py2.py3-none-any.whl
Collecting python-dateutil (from matplotlib==1.5.3)
  Downloading python_dateutil-2.6.0-py2.py3-none-any.whl (194kB)
    100% |██████████| 194KB 72kB/s
Collecting pytz (from matplotlib==1.5.3)
  Downloading pytz-2016.7-py2.py3-none-any.whl (480kB)
    100% |██████████| 481KB 546kB/s
Collecting pyparsing!=2.0.0,!=2.0.4,!=2.1.2,>=1.5.6 (from matplotlib==1.5.3)
  Downloading pyparsing-2.1.10-py2.py3-none-any.whl (56kB)
    100% |██████████| 61kB 491kB/s
Collecting numpy>=1.6 (from matplotlib==1.5.3)
  Downloading numpy-1.11.2.tgz (4.2MB)
    100% |██████████| 4.2MB 109kB/s
Collecting six (from cycler->matplotlib==1.5.3)
  Downloading six-1.10.0-py2.py3-none-any.whl
Building wheels for collected packages: numpy
  Running setup.py bdist_wheel for numpy ... error

  build_src
  building py_modules sources
  building library "npymath" sources
  No module named 'numpy.distutils._msvccompiler' in numpy.distutils; trying from distutils
  error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual C++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools

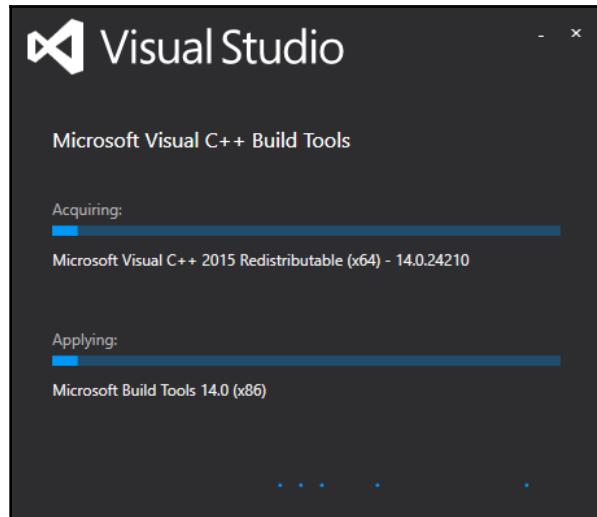
-----
Command "<:python3>python.exe -u -c "import setuptools, tokenize; file = __file__; exec(compile(open(file).read().replace('\r\n', '\n'), file, 'exec'))" install --record C:\Users\Burkhard\AppData\Local\Temp\pip-uqm4em-record\install-record.txt --single-version-externally-managed --compile" failed with error code 1 in C:\Users\Burkhard\AppData\Local\Temp\pip-build-hd94r1\numpy\

C:\Users\Burkhard\Desktop\2nd EDITION PACKT PYTHON GUI COOKBOOK\SW DOWNLOADS>
```

3. If you run into the preceding error, download Microsoft Visual C++ Build Tools and install them from <https://visualstudio.microsoft.com/visual-cpp-build-tools/>:



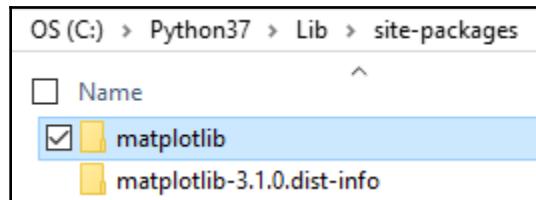
Starting the installation of Microsoft Visual C++ Build Tools appears as follows:



4. If you ran into the preceding error, rerun the Matplotlib installation using pip install:

```
PS C:\Users\Burkh\Desktop\2019 Third Edition Book> pip install matplotlib-3.1.0-cp37-cp37m-win_amd64.whl
Processing c:\users\burkh\desktop\2019 third edition book\matplotlib-3.1.0-cp37-cp37m-win_amd64.whl
Requirement already satisfied: kiwisolver>=1.0.1 in c:\python37\lib\site-packages (from matplotlib==3.1.0) (1.0.1)
Requirement already satisfied: python-dateutil>=2.1 in c:\python37\lib\site-packages (from matplotlib==3.1.0) (2.7.5)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,>=2.1.6,>=2.0.1 in c:\python37\lib\site-packages (from matplotlib==3.1.0) (2.3.0)
Requirement already satisfied: numpy>=1.11 in c:\python37\lib\site-packages (from matplotlib==3.1.0) (1.15.1)
Requirement already satisfied: cycler>=0.10 in c:\python37\lib\site-packages (from matplotlib==3.1.0) (0.10.0)
Requirement already satisfied: setuptools in c:\python37\lib\site-packages (from kiwisolver>=1.0.1->matplotlib==3.1.0) (39.0.1)
Requirement already satisfied: six>=1.5 in c:\python37\lib\site-packages (from python-dateutil>=2.1->matplotlib==3.1.0) (1.11.0)
Installing collected packages: matplotlib
  Found existing installation: matplotlib 3.0.2
    Uninstalling matplotlib-3.0.2:
      Successfully uninstalled matplotlib-3.0.2
Successfully installed matplotlib-3.1.0
PS C:\Users\Burkh\Desktop\2019 Third Edition Book>
```

5. Verify successful installation by looking into the site-packages folder:



Let's now go behind the scenes to understand the installation better.

How it works...

After downloading the wheel installer, we can now use pip to install the Matplotlib wheel.

In *Step 1*, make sure you download and install the Matplotlib version that matches the Python version you are using. For example, download and install `matplotlib-3.1.0-cp37-cp37m-win_amd64.whl` if you have Python 3.7 installed on a 64-bit OS, such as Microsoft Windows 10.



`amd64` in the middle of the executable name means you are installing the 64-bit version. If you are using a 32-bit x86 system, then installing `amd64` will not work. Similar problems can occur if you have installed a 32-bit version of Python and download 64-bit Python modules.

Depending upon what you have already installed on your system, running the `pip install matplotlib-3.1.0-cp37-cp37m-win_amd64.whl` command might start fine, but then it might not run to completion. Refer to the preceding screenshot during *Step 2* of what might happen during the installation. The installation ran into an error. The way to resolve this is to download and install **Microsoft Visual C++ Build Tools**, and we do this in *Step 3* from the website that is mentioned in the error for *Step 2* (<https://visualstudio.microsoft.com/visual-cpp-build-tools/>).



If you run into any issues installing Microsoft Visual C++ Build Tools, here is a helpful answer from Stack Overflow: <https://stackoverflow.com/a/54136652>. And here is a link to MS: <https://devblogs.microsoft.com/cppblog/announcing-visual-c-build-tools-2015-standalone-c-tools-for-build-environments/>.

After we have successfully installed the build tools, we can now rerun our Matplotlib installation to completion in *Step 4*. Just type in the same `pip install` command we have used previously in *Step 2*.

We can verify that we have successfully installed Matplotlib by looking at our Python installation directory, which we do in *Step 5*. After successful installation, the Matplotlib folder is added to `site-packages`. Depending upon where we installed Python, the full path to the `site-packages` folder on Windows can be `.. \Python37\Lib\site-packages`.

If you see the `matplotlib` folder added to the `site-packages` folder in your Python installation, then you have successfully installed Matplotlib.



Installing Python modules using `pip` is usually very easy, although you might run into some unexpected troubles. Follow the preceding steps and your installation will succeed.

Let's move on to the next recipe.

Creating our first chart

Now that we have all the required Python modules installed, we can create our own charts using Matplotlib.

We can create charts with only a few lines of Python code.

Getting ready

Successfully installing Matplotlib, as shown in the previous recipe, is a requirement for this recipe.

How to do it...

Using the minimum amount of code, we can create our first Matplotlib chart.

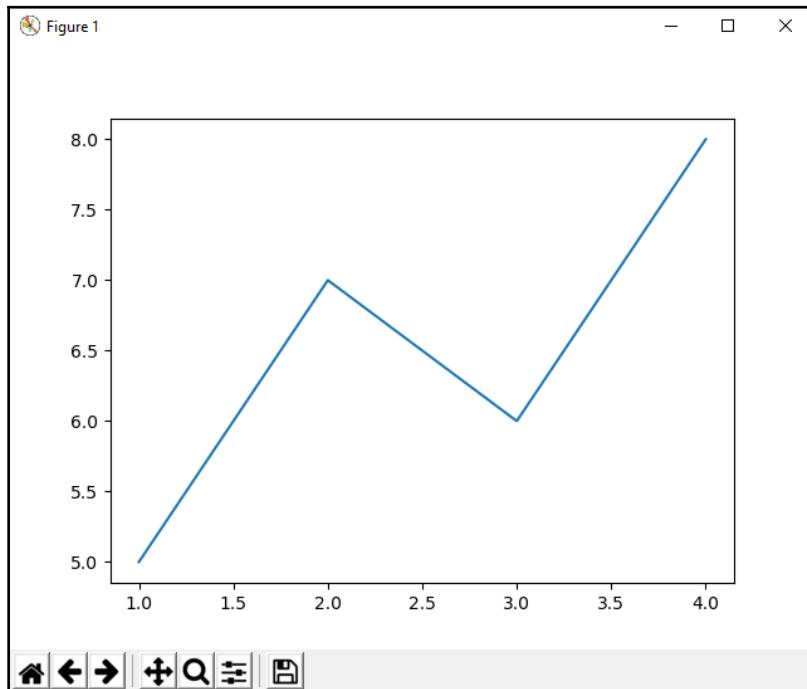
For the first chart, the steps are as follows:

1. Create a new Python module and save it as `Matplotlib_our_first_chart.py`.
2. Type the following code into the module:

```
import matplotlib.pyplot as plt
from pylab import show

x_values = [1,2,3,4]
y_values = [5,7,6,8]
plt.plot(x_values, y_values)
show()
```

3. Run the code to see the following chart:



Let's now go behind the scenes to understand the code better.

How it works...

First, we are importing `matplotlib.pyplot` and we alias it as `plt`. We then create two lists for our `x` and `y` values. We then pass the two lists into the `plt` or `plot` function.

We also import `show` from `pylab` and call it to display our chart.

Notice how this automatically creates a GUI for us that even comes with a number of buttons.

Play around with the buttons in the bottom-left corner because they are fully functional.



Also notice how the x and y axes scale automatically to display the data range of our x and y values.

There's more...

The Python Matplotlib module, combined with add-ons such as numpy, creates a very rich programming environment that enables us to perform mathematical computations and plot them in visual charts with ease.

Now, let's move on to the next recipe.

Placing labels on charts

So far, we have used the default Matplotlib GUI. Now, we will create some tkinter GUIs from which we will be using Matplotlib.

This will require a few more lines of Python code and the importing of some more libraries, and it is well worth the effort, because we are gaining control of our paintings using canvases.

We will position labels onto both the horizontal and the vertical axes, that is, x and y . We will do this by creating a Matplotlib figure that we will draw on.

You will also learn how to use subplots, which will enable you to draw more than one graph in the same GUI window.

Getting ready

With the necessary Python modules installed and knowing where to find the official online documentation and tutorials, we can now carry on with our creation of Matplotlib charts.

How to do it...

While `plot` is the easiest way to create a Matplotlib chart, using `Figure` in combination with `Canvas` creates a more custom-made graph, which looks much better and also enables us to add buttons and other widgets to it:

1. Create a new Python module and save it as `Matplotlib_labels.py`.
2. Type the following code into the module:

```
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import tkinter as tk
#-----
fig = Figure(figsize=(12, 8), facecolor='white')
#-----
# axis = fig.add_subplot(111) # 1 row, 1 column, only graph #<-- uncomment
axis = fig.add_subplot(211) # 2 rows, 1 column, Top graph
#-----
```

3. Add the following code under the preceding code:

```
xValues = [1,2,3,4]
yValues = [5,7,6,8]
axis.plot(xValues, yValues)
axis.set_xlabel('Horizontal Label')
axis.set_ylabel('Vertical Label')
# axis.grid()                                     # default line style
axis.grid(linestyle='--')                      # solid grid lines
```

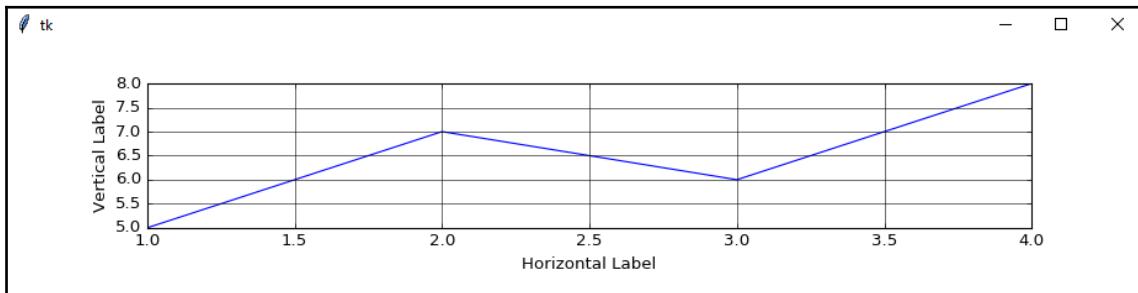
4. Next, add the following code under the preceding code:

```
#-----
def _destroyWindow():
    root.quit()
    root.destroy()
#-----
root = tk.Tk()
root.protocol('WM_DELETE_WINDOW', _destroyWindow)
#-----
```

5. Now, add the following code under the preceding code:

```
canvas = FigureCanvasTkAgg(fig, master=root)
canvas._tkcanvas.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
#-----
root.mainloop()
```

6. Running the preceding code results in the following chart:

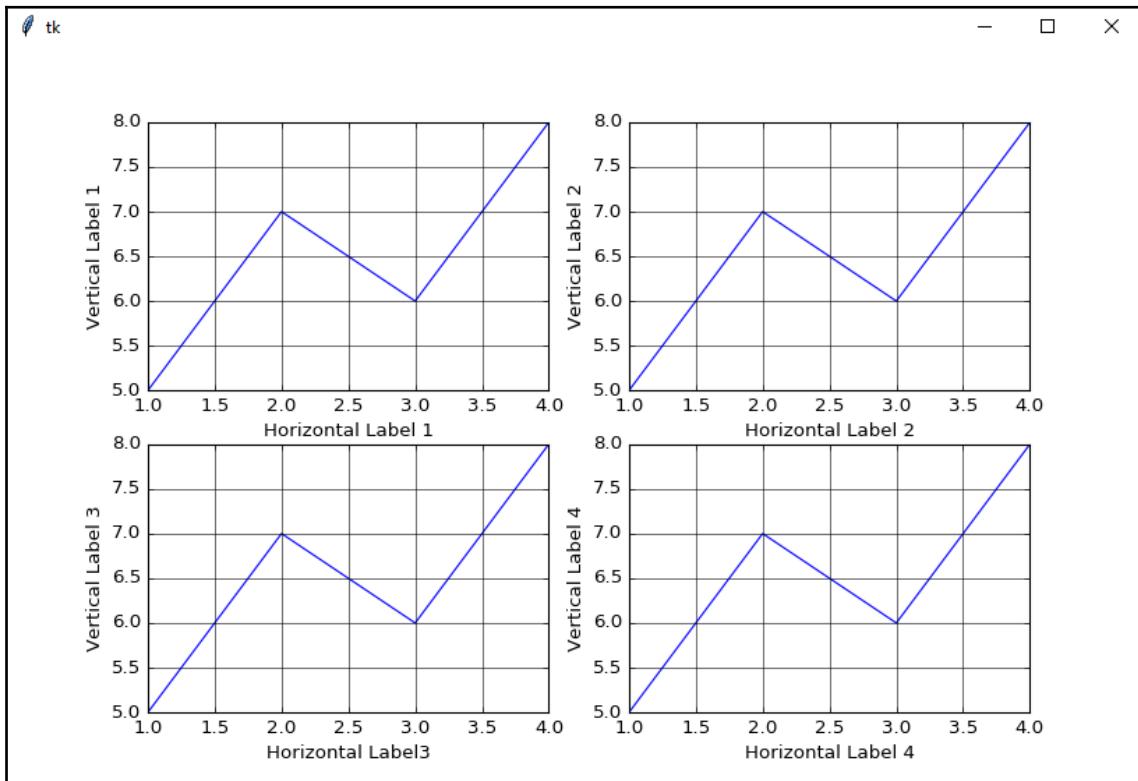


Now, let's deal with a new module:

1. Create a new module and save it as `Matplotlib_labels_four.py`.
2. Type the following new code into the module:

```
# imports and figure are the same as in the previous code
#-----
axis1 = fig.add_subplot(221)
axis2 = fig.add_subplot(222, sharex=axis1, sharey=axis1)
axis3 = fig.add_subplot(223, sharex=axis1, sharey=axis1)
axis4 = fig.add_subplot(224, sharex=axis1, sharey=axis1)
#-----
axis1.plot(xValues, yValues)
axis1.set_xlabel('Horizontal Label 1')
axis1.set_ylabel('Vertical Label 1')
axis1.grid(linestyle='--') # solid grid lines
#-----
axis2.plot(xValues, yValues)
axis2.set_xlabel('Horizontal Label 2')
axis2.set_ylabel('Vertical Label 2')
axis2.grid(linestyle='--') # solid grid lines
#-----
axis3.plot(xValues, yValues)
axis3.set_xlabel('Horizontal Label3')
axis3.set_ylabel('Vertical Label 3')
axis3.grid(linestyle='--') # solid grid lines
#-----
axis4.plot(xValues, yValues)
axis4.set_xlabel('Horizontal Label 4')
axis4.set_ylabel('Vertical Label 4')
axis4.grid(linestyle='--') # solid grid lines
#-----
# root and canvas are the same as in the previous code
```

3. Running the code results in the following chart being created:



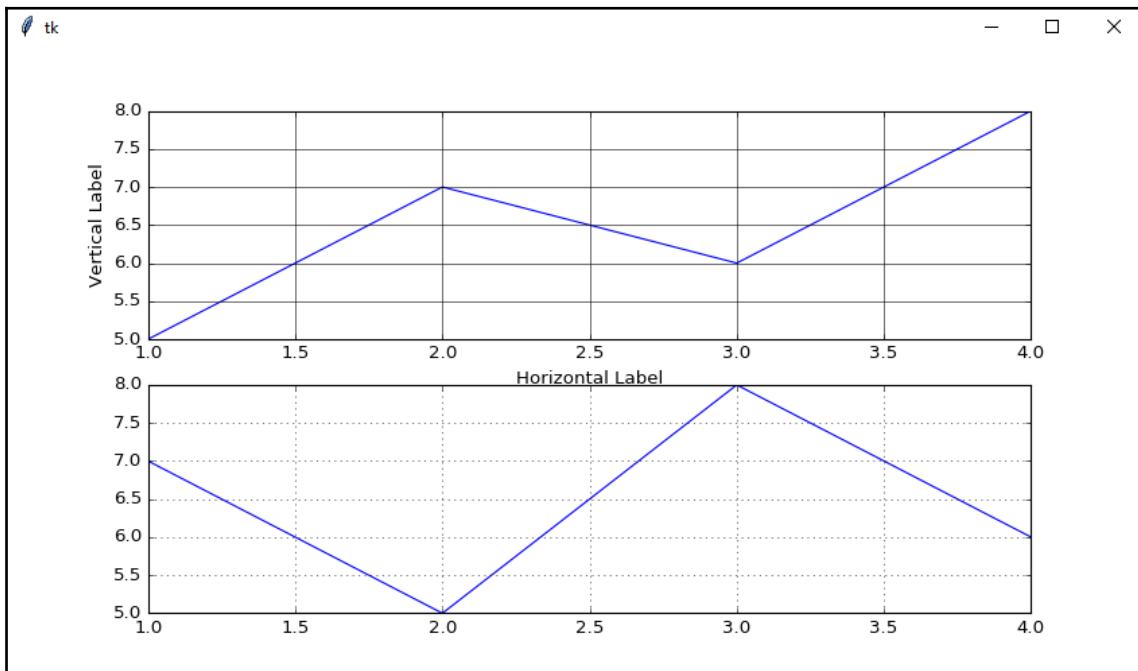
We can add more subplots by assigning them to the second position using `add_subplot(212)`:

1. Create a new module and save it as `Matplotlib_labels_two_charts.py`.
2. Type the following code into the module:

```
# imports and figure are the same as in the previous code
#-----
#-----axis = fig.add_subplot(211) # 2 rows, 1 column, Top graph
#-----
xValues = [1,2,3,4]
yValues = [5,7,6,8]
axis.plot(xValues, yValues)
axis.set_xlabel('Horizontal Label')
axis.set_ylabel('Vertical Label')
axis.grid(linestyle='--')      # solid grid lines
```

```
#-----  
axis1 = fig.add_subplot(212) # 2 rows, 1 column, Bottom graph  
#-----  
xValues1 = [1,2,3,4]  
yValues1 = [7,5,8,6]  
axis1.plot(xValues1, yValues1)  
axis1.grid() # default line style  
#-----  
#-----  
# root and canvas are the same as in the previous code
```

3. Run the code to see the following chart:



Let's now go behind the scenes to understand the code better.

How it works...

In the first line of code in `Matplotlib_labels.py`, in *Step 2*, after the import statements, we create an instance of a `Figure` object.



Here is a link to the official documentation: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure.add_subplot.

Next, we add subplots to this figure by calling `add_subplot(211)`.



The first number in `211` tells the figure *how many plots* to add, the second number determines *the number of columns*, and the third tells the figure *the order in which to display* the plots.

In *Step 3*, we create values, plot them, and we also add a grid and change its default line style.

Even though we only display one plot in the chart, by choosing `2` for the number of subplots, we are moving the plot up, which results in extra white space at the bottom of the chart. This first plot now only occupies 50% of the screen, which affects how large the grid lines of this plot are when being displayed.



Experiment with the code by uncommenting the code for `axis =` and `axis.grid()` to see the different effects. You also have to comment out the original line below each of them.

In *Step 4*, we create a callback function that correctly exits the `tkinter` GUI when the red X button gets clicked. We create an instance of `tkinter` and assign the callback to the `root` variable.

In *Step 5*, we create a canvas and use the `pack` geometry manager and, after that, we start the main windows GUI event loop.

Running the entire code in *Step 6* then creates the chart.

We can place more than one chart onto the same canvas. In `Matplotlib_labels_four.py`, most of the code is the same as in `Matplotlib_labels.py`. We are creating four axes and positioning them in two rows.



The important thing to observe is that we create one axis, which is then used as the shared x and y axes for the other graphs within the chart. In this way, we can achieve a database-like layout of the chart.

In `Matplotlib_labels_two_charts.py`, running the code now adds `axis1` to the chart. For the grid of the bottom plot, we left the line style at its default. The main difference compared with the previous charts is that we assigned the second chart to the second position using `add_subplot(212)`.

This means: 2 rows, 1 column, position 2 for this chart, which places it in the second row as there is only one column.

Now, let's move on to the next recipe.

How to give the chart a legend

Once we start plotting more than one line of data points, things might become a little bit unclear. By adding a legend to our graphs, we can identify data, and tell what it actually means.



We do not have to choose different colors to represent the different data. Matplotlib automatically assigns a different color to each line of the data points.

All we have to do is create the chart and add a legend to it.

Getting ready

In this recipe, we will enhance the chart from the previous recipe, *Placing labels on charts*. We will only plot one chart.

How to do it...

First, we will plot more lines of data on the same chart, and then we will add a legend to the chart.

1. Create a new module and save it as `Matplotlib_chart_with_legend.py`.

2. Type the following code into the module:

```
from matplotlib.figure import Figure
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import tkinter as tk
#-----
fig = Figure(figsize=(12, 5), facecolor='white')
#-----
```

3. Add the following code under the preceding code:

```
axis = fig.add_subplot(111) # 1 row, 1 column

xValues = [1,2,3,4]
yValues0 = [6,7.5,8,7.5]
yValues1 = [5.5,6.5,8,6]
yValues2 = [6.5,7,8,7]

t0, = axis.plot(xValues, yValues0)
t1, = axis.plot(xValues, yValues1)
t2, = axis.plot(xValues, yValues2)

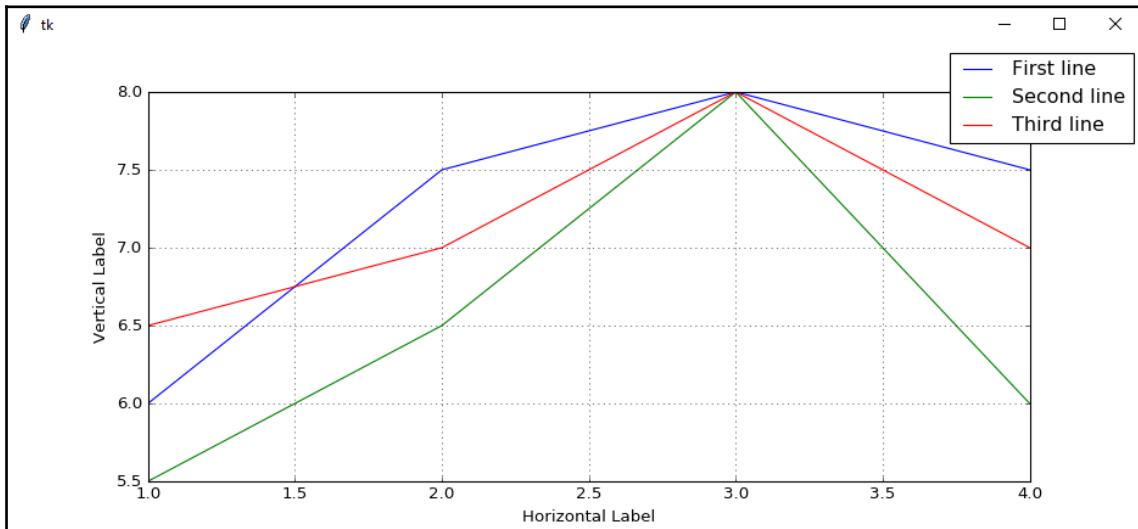
axis.set_ylabel('Vertical Label')
axis.set_xlabel('Horizontal Label')
axis.grid()

fig.legend((t0, t1, t2), ('First line', 'Second line', 'Third
line'), 'upper right')
#-----
```

4. Next, add the following code under the preceding code:

```
def _destroyWindow():
    root.quit()
    root.destroy()
#-----
root = tk.Tk()
root.protocol('WM_DELETE_WINDOW', _destroyWindow)
#-----
canvas = FigureCanvasTkAgg(fig, master=root)
canvas._tkcanvas.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
#-----
root.mainloop()
```

5. Running the code creates the following chart, which has a legend in the upper-right corner:

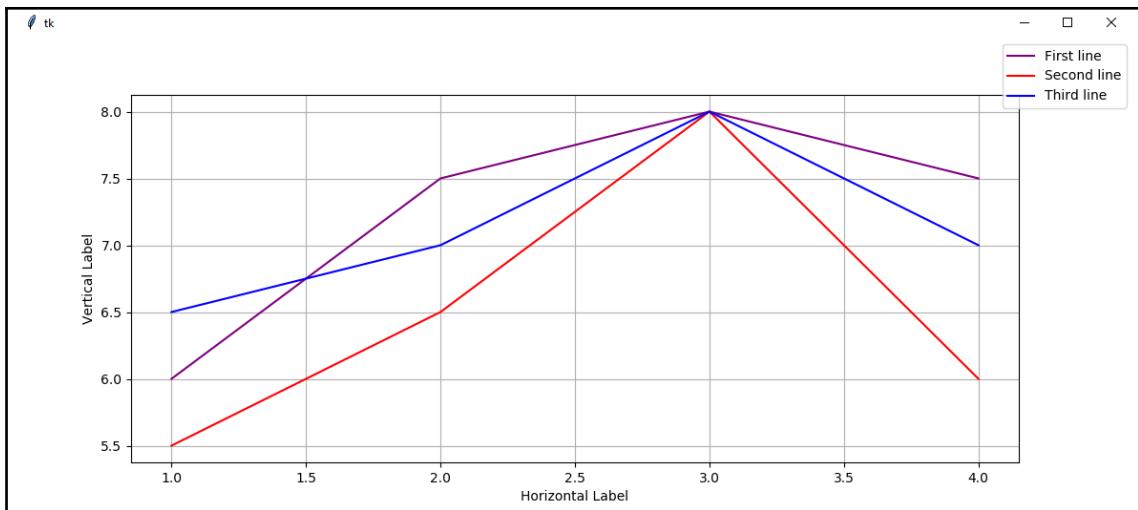


Next, we change the default colors of the lines in the legend.

1. Open `Matplotlib_chart_with_legend.py` and save it as `Matplotlib_chart_with_legend_colors.py`.
2. Add the following colors to each plot:

```
t0, = axis.plot(xValues, yValues0, color = 'purple')
t1, = axis.plot(xValues, yValues1, color = 'red')
t2, = axis.plot(xValues, yValues2, color = 'blue')
```

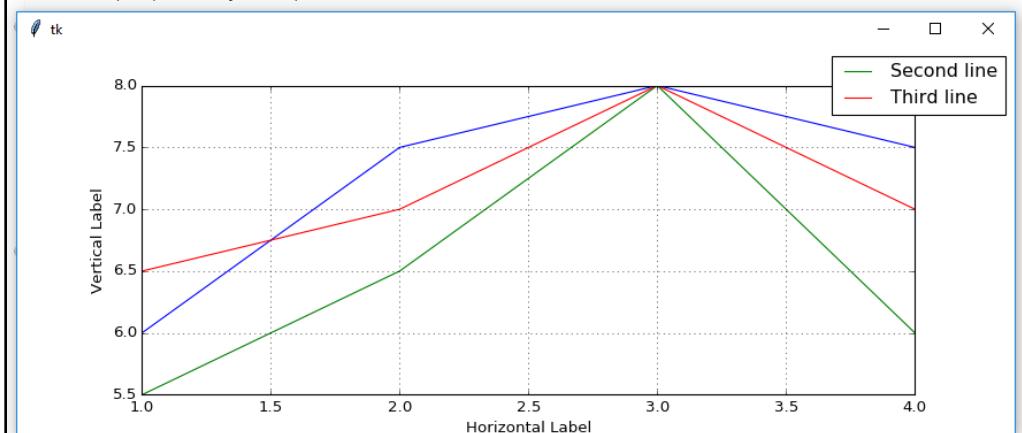
3. Run the modified code and observe the different colors:



Now, let's have a closer look at the correct syntax when assigning the plots to variables.

1. Open `Matplotlib_chart_with_legend.py` and save it as `Matplotlib_chart_with_legend_missing_comma.py`.
2. Remove the comma after `t0`.
3. Run the code.
4. Notice how `First line` no longer appears in the top-right legend:

```
# the commas after t0, t1 and t2 are required
t0 = axis.plot(xValues, yValues0)           # no comma here
t1, = axis.plot(xValues, yValues1)
t2, = axis.plot(xValues, yValues2)
```



Let's now go behind the scenes to understand the code better.

How it works...

In `Matplotlib_chart_with_legend.py`, we are only plotting one graph in this recipe.

For *Step 2*, refer to the explanation from the previous recipe, *Placing labels on charts*, as the code is the same except that we are slightly modifying the size of the figure via the `figsize` attribute.

In *Step 3*, we change `fig.add_subplot(111)` to use `111`. Next, we create three Python lists that contain the values to be plotted. When we plot the data, we save the references to the plots in local variables.

We create the legend by passing it in a tuple with the references to the three plots, and then pass it in another tuple that contains the strings that are then displayed in the legend, and, in the third argument, we position the legend within the chart.

For *Step 4*, refer to the explanation from the previous recipe, *Placing labels on charts*, as the code is the same.



You can find the official documentation for the `tkinter` protocol at this link: <https://www.tcl.tk/man/tcl8.4/TkCmd/wm.htm#M39>.

In *Step 5*, when running the code, we can see that our chart now has a legend for each of the three lines of data.

The default settings of `Matplotlib` assign a color scheme to the lines being plotted. In `Matplotlib_chart_with_legend_colors.py`, we can easily change this default setting of colors to the colors we prefer by setting an attribute when we plot each axis.

We do this in *Step 2* by using the `color` attribute and assigning it an available color value. Running the code in *Step 3* now shows different colors than the default colors.

In `Matplotlib_chart_with_legend_missing_comma.py`, we intentionally remove the comma after `t0` to see what effect this has.



Note that the comma after the variable assignments of `t0`, `t1`, and `t2` is not a mistake. It is required in order to create the legend.

The comma after each variable **unpacks** the list value into the variable. This value is a `Line2D` object of `Matplotlib`. If we leave the comma out, our legend will not be displayed because the `Line2D` object is embedded in a list and we have to unpack it out of the list.



When we remove the comma after the `t[0]` assignment, we get an error, and the first line no longer appears in the figure. The chart and legend still get created, but without the first line appearing in the legend.

Let's move on to the next recipe.

Scaling charts

In the previous recipes, while creating our first charts and enhancing them, we hardcoded the scaling of how those values are visually represented.

While this served us well for the values we were using, we might have to plot charts from large databases.

Depending on the range of that data, our hardcoded values for the vertical y -dimension might not always be the best solution, and may make it hard to see the lines in our charts.

Getting ready

We will improve our code from the previous recipe, *How to give the chart a legend*. If you have not typed in all of the code from the previous recipes, just download the code for this chapter from the Packt website, and it will get you started (and then you can have a lot of fun creating GUIs, charts, and so on using Python).

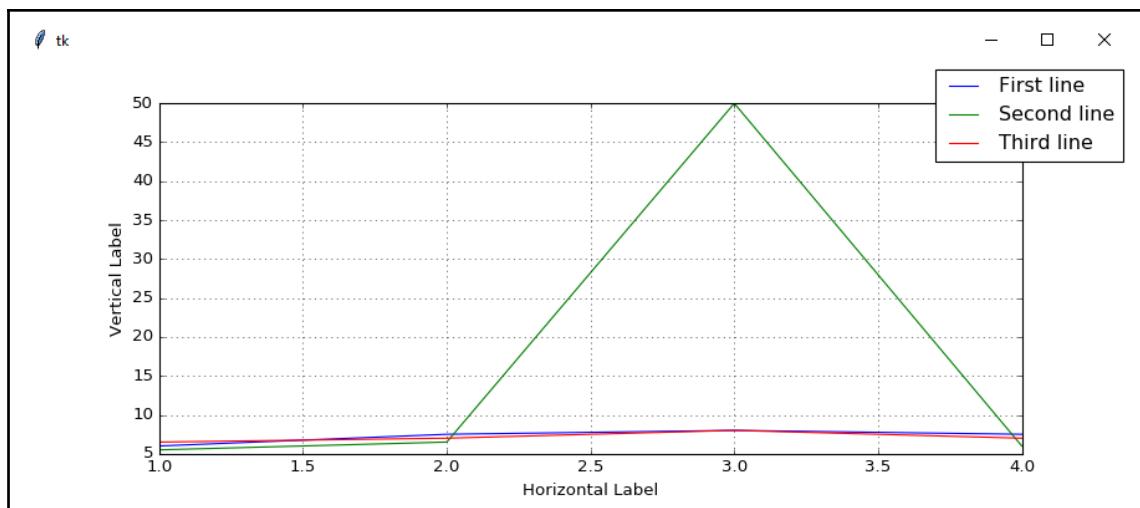
How to do it...

We will modify the `yValues1` line of code from the previous recipe to use 50 as the third value:

1. Open `Matplotlib_chart_with_legend.py` and save it as `Matplotlib_labels_two_charts_not_scaled.py`.
2. Change the third value in the list of `yValues1` to 50:

```
axis = fig.add_subplot(111)          # 1 row, 1 column
xValues = [1,2,3,4]
yValues0 = [6,7.5,8,7.5]
yValues1 = [5.5,6.5,50,6]           # one very high value (50)
yValues2 = [6.5,7,8,7]
```

3. Run the code to see the following chart:

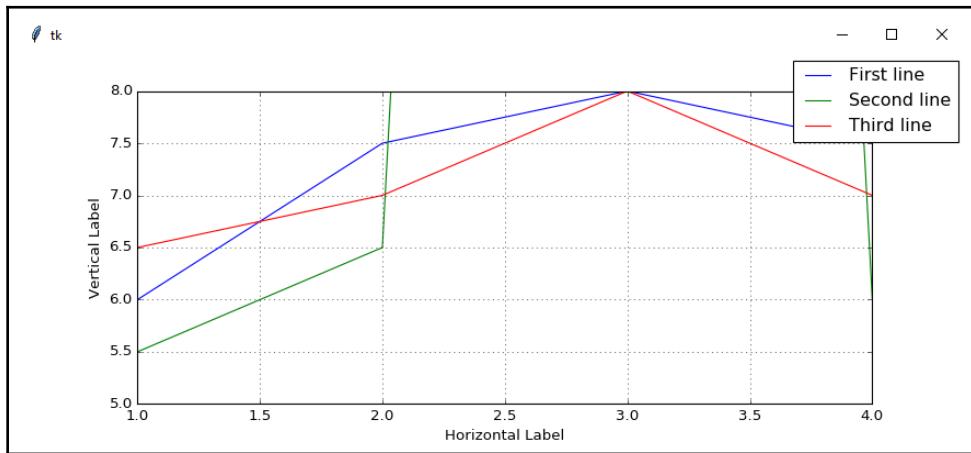


1. Open `Matplotlib_labels_two_charts_not_scaled.py` and save it as `Matplotlib_labels_two_charts_scaled.py`.
2. Add `axis.set_ylim(5, 8)` under the value code:

```
yValues0 = [6,7.5,8,7.5]
yValues1 = [5.5,6.5,50,6]           # one very high value (50)
yValues2 = [6.5,7,8,7]

axis.set_ylim(5, 8)                  # limit the vertical display
```

3. After running the code, the following chart appears:



Let's now go behind the scenes to understand the code better.

How it works...

In `Matplotlib_labels_two_charts_not_scaled.py`, the only difference to the code that created the chart in the previous recipe is a single data value.

By changing one value that is not close to the average range of all the other values for all plotted lines, the visual representation of the data has dramatically changed. We lost a lot of detail regarding the overall data, and we now mainly see one high spike.

So far, our charts have adjusted themselves according to the data they visually represent.

While this is a practical feature of Matplotlib, this is not always what we want. We can restrict the scale of the chart being represented by limiting the vertical y -dimension.

In `Matplotlib_labels_two_charts_scaled.py`, the `axis.set_ylim(5, 8)` line of code now limits the start value to 5 and the end value of the vertical display to 8.

Now, when we create our chart, the high value peak no longer has the impact it had before.

We increased one value in the data, which resulted in a dramatic effect. By setting limits to the vertical and horizontal displays of the chart, we can see the data we are most interested in.

Spikes, like the ones just shown, can be of great interest too. It all depends on what we are looking for. The visual representation of the data is of great value.



A picture is worth a thousand words.

Now, let's move on to the next recipe.

Adjusting the scale of charts dynamically

In the previous recipe, we learned how we can limit the scaling of our charts. In this recipe, we will go one step further by dynamically adjusting the scaling by both setting a limit and analyzing our data before we represent it.

Getting ready

We will enhance the code from the previous recipe, *Scaling charts*, by reading in the data we are plotting dynamically, averaging it, and then adjusting our chart.

While we would typically read in the data from an external source, in this recipe, we'll create the data we are plotting using Python lists, as can be seen in the code in the following section.

How to do it...

We are creating our own data in our Python module by assigning lists with data to the `xValues` and `yValues` variables. Let's now modify the code to set limits on both the *x* and *y* dimensions.

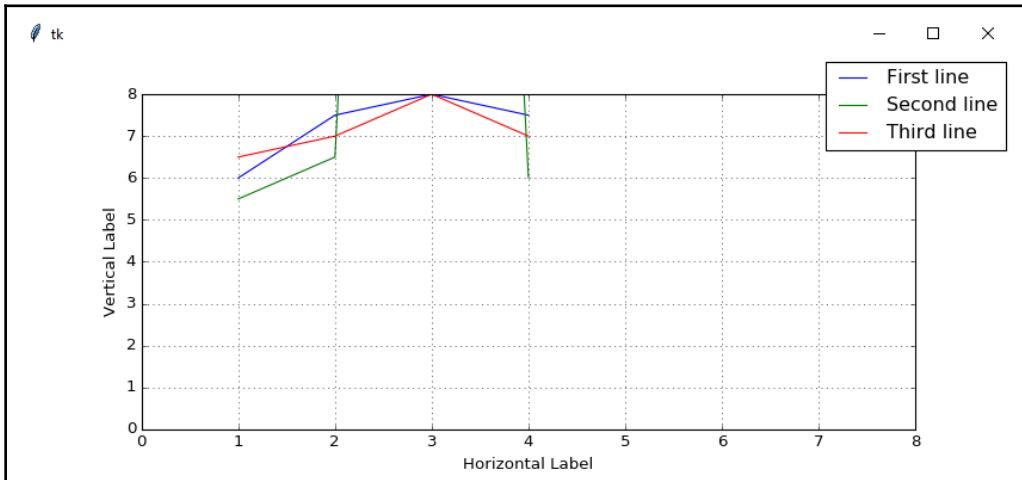
1. Open `Matplotlib_labels_two_charts_scaled.py` and save it as `Matplotlib_labels_two_charts_scaled_dynamic_spike.py`.
2. Add/adjust the `set_ylim` and `set_xlim` code as follows:

```
xValues = [1,2,3,4]

yValues0 = [6,7.5,8,7.5]
yValues1 = [5.5,6.5,50,6]           # one very high value (50)
yValues2 = [6.5,7,8,7]
```

```
axis.set_ylim(0, 8)                      # lower limit (0)
axis.set_xlim(0, 8)                       # use same limits for x
```

3. When we run the modified code, we get the following result:



Modify the code as follows:

1. Open `Matplotlib_labels_two_charts_scaled_dynamic_spike.py` and save it as `Matplotlib_labels_two_charts_scaled_dynamic.py`.
2. Insert the following new code starting with `yAll`:

```
xValues = [1,2,3,4]

yValues0 = [6,7.5,8,7.5]
yValues1 = [5.5,6.5,50,6]           # one very high value (50)
yValues2 = [6.5,7,8,7]
yAll = [yValues0, yValues1, yValues2] # list of lists

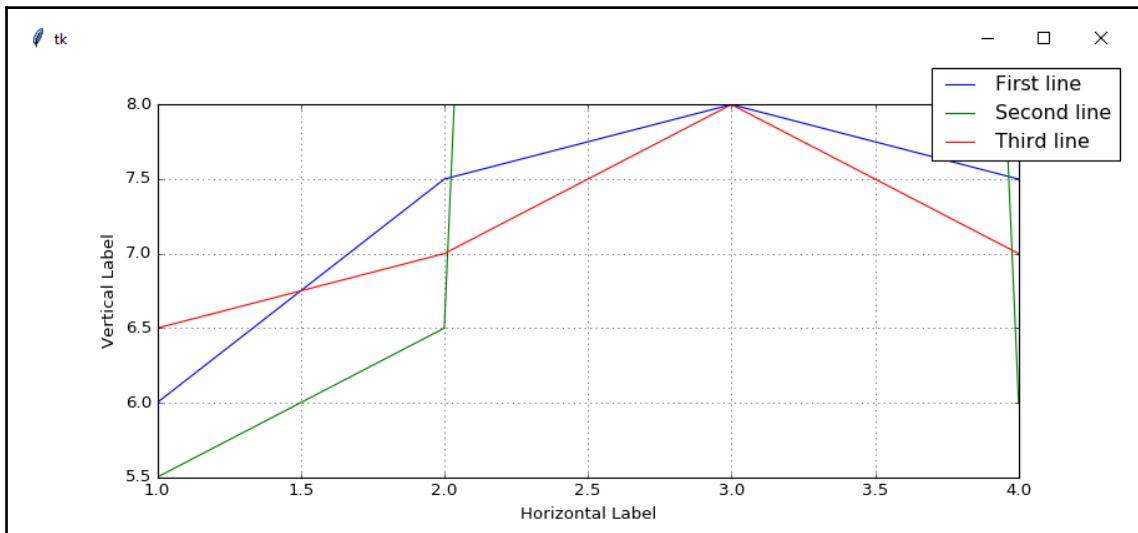
# flatten list of lists retrieving minimum value
minY = min([y for yValues in yAll for y in yValues])

yUpperLimit = 20
# flatten list of lists retrieving max value within defined limit
maxY = max([y for yValues in yAll for y in yValues if y <
yUpperLimit])

# dynamic limits
axis.set_ylim(minY, maxY)
axis.set_xlim(min(xValues), max(xValues))
```

```
t0, = axis.plot(xValues, yValues0)
t1, = axis.plot(xValues, yValues1)
t2, = axis.plot(xValues, yValues2)
```

3. Running the code results in the following chart:



Let's now go behind the scenes to understand the code better.

How it works...

In many graphs, the beginning of the x and y coordinate system starts at $(0, 0)$. This is usually a good idea, so we adjusted our chart coordinate code accordingly. In `Matplotlib_labels_two_charts_scaled_dynamic_spike.py`, we have set the same limits for x and y , hoping that our chart might look more balanced. Looking at the result, this is not the case.

Maybe starting at $(0, 0)$ was not such a great idea after all.



What we really want to do is to adjust our chart dynamically according to the range of the data, while, at the same time, restricting the values that are too high or too low.

We can do this by parsing all the data to be represented in the chart while, at the same time, setting some explicit limits. In `Matplotlib_labels_two_charts_scaled_dynamic.py`, we adjusted both its *x* and *y* dimensions dynamically. Note how the *y*-dimension starts at **5.5**. The chart also no longer starts at **(0, 0)**, giving us more valuable information about our data.

We are creating a list of lists for the *y*-dimension data and then using a list comprehension wrapped into a call to Python's `min()` and `max()` functions.



If list comprehensions seem to be a little bit advanced, what they basically are is a very compressed loop. They are also designed to be faster than a regular programming loop.

In the Python code, we created three lists that hold the *y*-dimensional data to be plotted. We then created another list that holds those three lists, which creates a list of lists, as follows:

```
yValues0 = [6, 7.5, 8, 7.5]
yValues1 = [5.5, 6.5, 50, 6]                      # one very high value (50)
yValues2 = [6.5, 7, 8, 7]
yAll = [yValues0, yValues1, yValues2]  # list of lists
```

We are interested in getting both the minimum value of all of the *y*-dimensional data as well as the maximum value contained within these three lists.

We can do this via a Python list comprehension:

```
# flatten list of lists retrieving minimum value
minY = min([y for yValues in yAll for y in yValues])
```

After running the list comprehension, `minY` is **5.5**.

The preceding line of code is the list comprehension that runs through all the values of all the data contained within the three lists and finds the minimum value using the Python `min` keyword.

In the very same pattern, we find the maximum value contained in the data we wish to plot. This time, we'll also set a limit within our list comprehension, which ignores all the values that exceed the limit we specified, as follows:

```
yUpperLimit = 20
# flatten list of lists retrieving max value within defined limit
maxY = max([y for yValues in yAll for y in yValues if y <
yUpperLimit])
```

After running the preceding code with our chosen restriction, `maxY` has the value of 8 (not 50).

We applied a restriction to the max value, according to a predefined condition, choosing 20 as the maximum value to be displayed in the chart.

For the *x*-dimension, we simply called `min()` and `max()` in the `Matplotlib` method to scale the limits of the chart dynamically.

In this recipe, we have created several `Matplotlib` charts and adjusted some of the many available attributes. We also used core Python to control the scaling of the charts dynamically.

6

Threads and Networking

In this chapter, we will extend the functionality of our Python GUI using threads, queues, and network connections.



A `tkinter` GUI is a single-threaded application. Every function that involves sleep or wait time has to be called in a separate thread; otherwise, the `tkinter` GUI freezes.

When we run our Python GUI, in Windows Task Manager, we can see that a new `python.exe` process has been launched. When we give our Python GUI a `.pyw` extension, then the process created will be `python.pyw`, which can be seen in Task Manager as well.

When a process is created, the process automatically creates a main thread to run our application. This is called a *single-threaded application*.



Single-threaded processes contain the execution of instructions in a single sequence. In other words, one command is processed at a time.

For our Python GUI, a single-threaded application will lead to our GUI becoming frozen as soon as we call a longer-running task, such as clicking a button that has a sleep time of a few seconds. In order to keep our GUI responsive, we have to use *multithreading*, and this is what we will study in this chapter.



Our GUI runs in a single thread. Knowing how to use multiple threads is an important concept for GUI development.

We can also create multiple processes by creating multiple instances of our Python GUI, as can be seen in Task Manager, where we can see several `python.exe` processes running at the same time.



Processes are isolated from each other by design and do not share common data. In order to communicate between separate processes, we have to use **Inter-Process Communication (IPC)**, which is an advanced technique. Threads, on the other hand, do share common data, code, and files, which makes communication between threads within the same process much easier than when using IPC. A great explanation of threads can be found at https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html.

In this chapter, we will learn how to keep our Python GUI responsive and keep it from freezing. Having this knowledge is essential when creating working GUIs, and knowing how to create threads and use queues increases your programming skills.

We will also use TCP/IP to connect our GUI to a network. In addition to that, we will read a URL web page, which is also a networking component on the internet.

Here is the overview of Python modules for this chapter:

Ch06_Code
> Backup
> _init_.py
> GUI_copy_files_limit.py
> GUI_copy_files.py
> GUI_multiple_threads_sleep_freeze.py
> GUI_multiple_threads_starting_a_thread.py
> GUI_multiple_threads_stopping_a_thread.py
> GUI_multiple_threads_thread_in_method.py
> GUI_multiple_threads.py
> GUI_passing_queues_member.py
> GUI_passing_queues.py
> GUI_queues_put_get_loop_endless_threaded.py
> GUI_queues_put_get_loop_endless.py
> GUI_queues_put_get_loop.py
> GUI_queues_put_get.py
> GUI_queues.py
> GUI_TCP_IP.py
> GUI_URL.py
> pyc.ico
> Queues.py
> TCP_Server.py
> ToolTip.py
> URL.py

We will create threads, queues, and TCP/IP sockets using Python 3.7 or later.

To sum it up, we will cover the following recipes:

- How to create multiple threads
- Starting a thread
- Stopping a thread
- How to use queues
- Passing queues among different modules
- Using dialog widgets to copy files to your network
- Using TCP/IP to communicate via networks
- Using `urlopen` to read data from websites

How to create multiple threads

Multiple threads are necessary in order to keep our GUI responsive. Without running of our GUI program using multiple threads, our application will freeze and possibly crash.

Getting ready

Multiple threads run within the same computer process memory space. There is no need for IPC, which would complicate our code. In this recipe, we will avoid IPC by using threads.

How to do it...

First, we will increase the size of our `ScrolledText` widget, making it larger. Let's increase `scrol_w` to 40 and `scrol_h` to 10.

We will start by using the latest code from chapter 5, *Matplotlib Charts*:

1. Open `Ch04_Code.GUI_OOP_class_imported_tooltip.py` and save it as `GUI_multiple_threads.py`.

2. Make the changes shown in the following code:

```
# Using a scrolled Text control
scrol_w = 40; scrol_h = 10      # increase sizes
self.scrol = scrolledtext.ScrolledText(mighty, width=scrol_w,
height=scrol_h, wrap=tk.WORD)
self.scrol.grid(column=0, row=3, sticky='WE', columnspan=3)
```

3. Modify `self.spin.grid` to use `sticky`:

```
# Adding a Spinbox widget
self.spin = Spinbox(mighty, values=(1, 2, 4, 42, 100), width=5,
bd=9, command=self._spin)
self.spin.grid(column=0, row=2, sticky='W') # align left, use
sticky
```

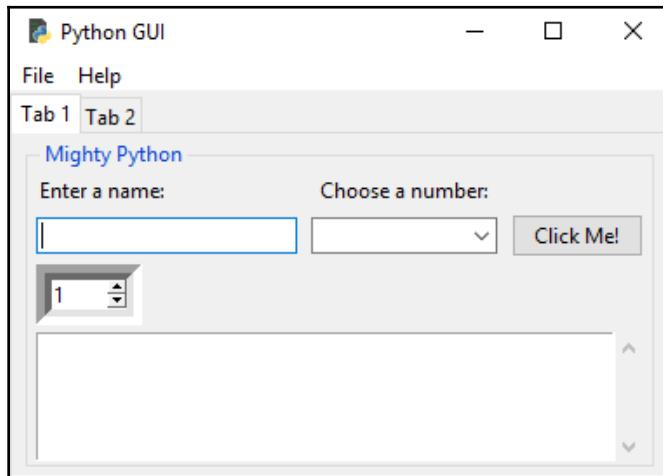
4. Increase the width size of Entry:

```
# Adding a Textbox Entry widget
self.name = tk.StringVar()
self.name_entered = ttk.Entry(mighty, width=24,
# increase width
textvariable=self.name)
self.name_entered.grid(column=0, row=1, sticky='W')
```

5. Increase the width size of Combobox to 14:

```
ttk.Label(mighty, text="Choose a number:").grid(column=1, row=0)
number = tk.StringVar()
self.number_chosen = ttk.Combobox(mighty, width=14,
# increase width
textvariable=number, state='readonly')
self.number_chosen['values'] = (1, 2, 4, 42, 100)
self.number_chosen.grid(column=1, row=1)
self.number_chosen.current(0)
```

6. Run the code and observe the output:



7. Import Thread from Python's built-in threading module:

```
#=====
# imports
#=====
import tkinter as tk
...
from threading import Thread
```

8. Add the `method_in_a_thread` method:

```
class OOP():
    def method_in_a_thread(self):
        print('Hi, how are you?')
```

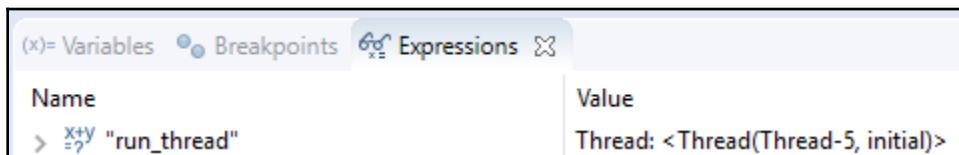
9. Create a thread as follows:

```
#=====
# Start GUI
#=====
oop = OOP()

# Running methods in Threads
run_thread = Thread(target=oop.method_in_a_thread) # create Thread

oop.win.mainloop()
```

10. Set a breakpoint or use a print statement for the `run_thread` variable:



Let's go behind the scenes to understand the code better.

How it works...

After our first changes in `step 2` to `GUI_multiple_threads.py`, when we run the resulting GUI, the `Spinbox` widget is center-aligned in relation to the `Entry` widget above it, which does not look good. We'll change this by left-aligning the widget. We add `sticky='W'` to the `grid` control to left-align the `Spinbox` widget.

The GUI could still look better, so next, we increase the size of the `Entry` widget to get a more balanced GUI layout. After that, we also increase the `Combobox` widget. Running the modified and improved code results in a larger GUI, which we will use for this recipe and for the following recipes.

In order to create and use threads in Python, we have to import the `Thread` class from the `threading` module. After adding the `method_in_a_thread` method, we can now call our threaded method in the code, saving the instance in a variable called `run_thread`.

Now we have a method that is threaded, but when we run the code, nothing gets printed to the console!



We have to start the thread first before it can run, and the next recipe will show us how to do this.

However, setting a breakpoint after the GUI main event loop proves that we did indeed create a thread object, as can be seen in the Eclipse IDE debugger.

In this recipe, we prepared our GUI to use threads by first increasing the GUI size so we can see the results printed to the `ScrolledText` widget in a better way. We then imported the `Thread` class from the Python `threading` module. Next, we created a method that we call in a thread from within our GUI.

Let's move on to the next recipe.

Starting a thread

This recipe will show us how to start a thread. It will also demonstrate why threads are necessary to keep our GUI responsive during long-running tasks.

Getting ready

Let's first see what happens when we call a function or a method of our GUI that has `sleep` associated with it without using threads.



We are using `sleep` here to simulate a real-world application that might have to wait for a web server or database to respond, a large file transfer, or complex computations to complete its task. `sleep` is a very realistic placeholder and shows the principle involved.

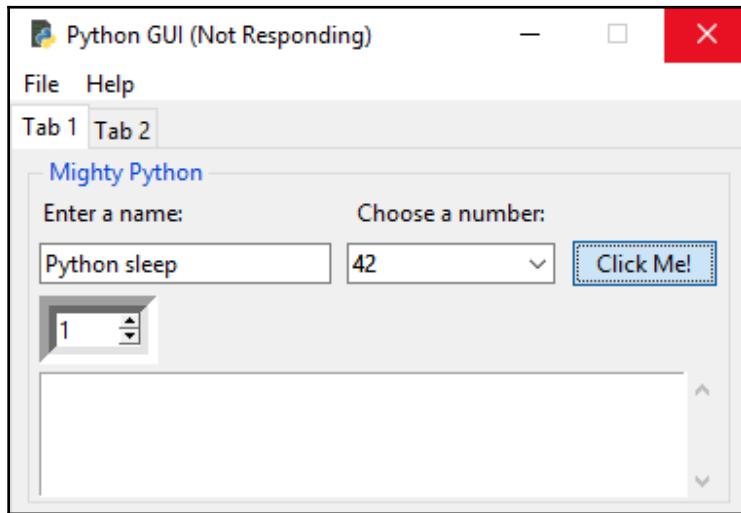
How to do it...

Adding a loop into our button callback method with some `sleep` time results in our GUI becoming unresponsive and, when we try to close the GUI, things get even worse.

1. Open `GUI_multiple_threads.py` and save it as `GUI_multiple_threads_sleep_freeze.py`.
2. Make the following changes to the code:

```
# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get() + ' '
+ self.number_chosen.get())
    # Non-threaded code with sleep freezes the GUI
    for idx in range(10):
        sleep(5)
        self.scrol.insert(tk.INSERT, str(idx) + 'n')
```

3. Running the preceding code results in the following screenshot:



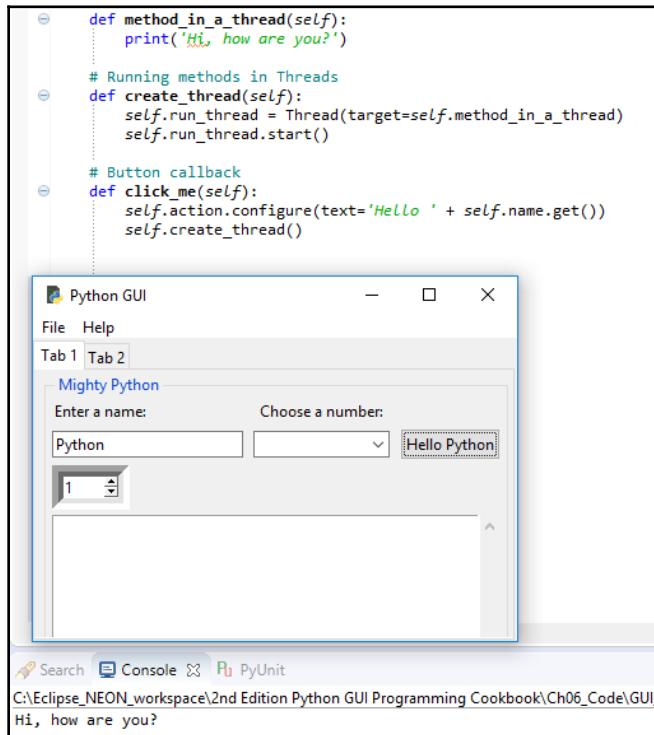
4. Let's move the creation of the thread into its own method and then call this method from the button callback method:

1. Open `GUI_multiple_threads_sleep_freeze.py` and save it as `GUI_multiple_threads_starting_a_thread.py`.
2. Add the following code:

```
# Running methods in Threads
def create_thread(self):
    self.run_thread =
        Thread(target=self.method_in_a_thread)
    self.run_thread.start() # start the thread

# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get())
    self.create_thread()
```

5. Run the code and observe the output. Running the code now no longer freezes our GUI:

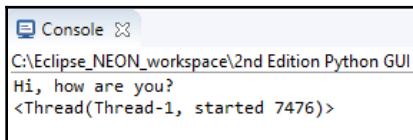


We can print the instance of the thread by following these steps:

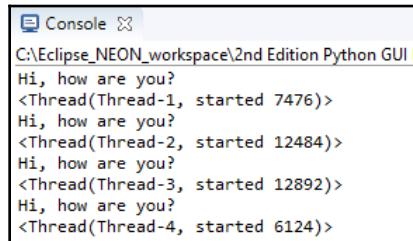
1. Open `GUI_multiple_threads_starting_a_thread.py`.
2. Add a `print` statement to the code:

```
# Running methods in Threads
def create_thread(self):
    self.run_thread =
        Thread(target=self.method_in_a_thread)
    self.run_thread.start()                      # start the thread
    print(self.run_thread)
```

6. Clicking the button now creates the following printout:



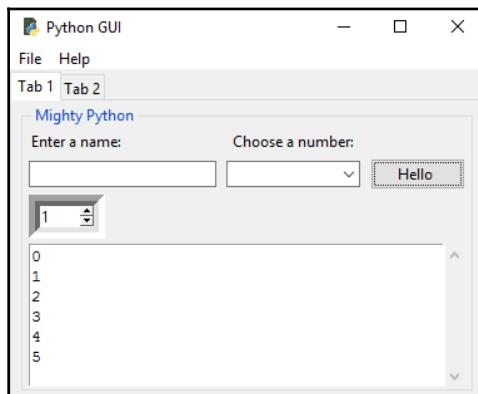
7. On clicking the button several times, you get the following output:



8. Move the code with `sleep` into a loop in the `method_in_a_thread` method:

```
def method_in_a_thread(self):
    print('Hi, how are you?')
    for idx in range(10):
        sleep(5)
        self.scrol.insert(tk.INSERT, str(idx) + '\n')
```

9. Click the button, change tabs, and then click on other widgets:



Let's go behind the scenes to understand the code better.

How it works...

In `GUI_multiple_threads_sleep_freeze.py`, we added a `sleep` statement and noticed how our GUI became unresponsive.



If we wait long enough, the method will eventually complete, but during this time, none of our GUI widgets respond to click events. We solve this problem by using threads.

Unlike regular Python functions and methods, we have to `start` a method that will be run in its own thread! This is what we did next in

`GUI_multiple_threads_starting_a_thread.py`. Clicking the button now results in the `create_thread` method being called, which, in turn, calls the `method_in_a_thread` method.

First, we create a thread and target it at a method. Next, we start the thread that runs the targeted method in a new thread. Running the code now no longer freezes our GUI.



The GUI itself runs in its own thread, which is the main thread of the application.

When we click the button several times, we can see that each thread gets assigned a unique name and ID. After moving the code with `sleep` into a loop in the `method_in_a_thread` method, we are able to verify that threads really do solve our problem.

When clicking the button, while the numbers are being printed into the `ScrolledText` widget with a five-second delay, we can click around anywhere in our GUI, switch tabs, and so on. Our GUI has become responsive again because we are using threads!

In this recipe, we called the methods of our GUI class in their own threads and learned that we have to start the threads. Otherwise, the thread gets created but just sits there waiting for us to run its target method. Also, we noticed that each thread gets assigned a unique name and ID. And finally, we simulated long-running tasks by inserting a `sleep` statement into our code, which showed us that threads can indeed solve our problem.

Let's move on to the next recipe.

Stopping a thread

We have to start a thread to actually make it do something by calling the `start()` method so, intuitively, we expect there to be a matching `stop()` method, but there is no such thing. In this recipe, we will learn how to run a thread as a background task, which is called a *daemon*. When closing the main thread, which is our GUI, all daemons will automatically be stopped as well.

Getting ready

When we call methods in a thread, we can also pass arguments and keyword arguments to the method. We start this recipe by doing exactly that. We will start with the code from the previous recipe.

How to do it...

By adding `args=[8]` to the thread constructor and modifying the targeted method to expect arguments, we can pass arguments to the threaded methods. The parameter to `args` has to be a sequence, so we will wrap our number in a Python list. Let's go through the process:

1. Open `GUI_multiple_threads_starting_a_thread.py` and save it as `GUI_multiple_threads_stopping_a_thread.py`.
2. Change `run_thread` to `self.run_thread` and `arg=[8]`:

```
# Running methods in Threads
def create_thread(self):
    self.run_thread = Thread(target=self.method_in_a_thread,
                            args=[8])
    self.run_thread.start()
    print(self.run_thread)
    print('createThread():', self.run_thread.isAlive())
```

3. Add `num_of_loops` as a new argument to `method_in_a_thread`:

```
def method_in_a_thread(self, num_of_loops=10):
    for idx in range(num_of_loops):
        sleep(1)
        self.scrol.insert(tk.INSERT, str(idx) + 'n')
    sleep(1)
    print('method_in_a_thread():', self.run_thread.isAlive())
```

4. Run the code, click the button, and then close the GUI:

The screenshot shows the Eclipse IDE interface. On the left is the code editor with Python code. On the right is the 'Console' tab showing the execution output.

```
def method_in_a_thread(self, num_of_loops=10):
    print('Hi, how are you?')
    for idx in range(num_of_loops):
        sleep(1)
        self.scrol.insert(tk.INSERT, str(idx) + '\n')
    print('method_in_a_thread():', self.run_thread.isAlive())

# Running methods in Threads
def create_thread(self):
    self.run_thread = Thread(target=self.method_in_a_thread, args=[8])
    self.run_thread.start()
    print(self.run_thread)
    print('createThread():', self.run_thread.isAlive())
```

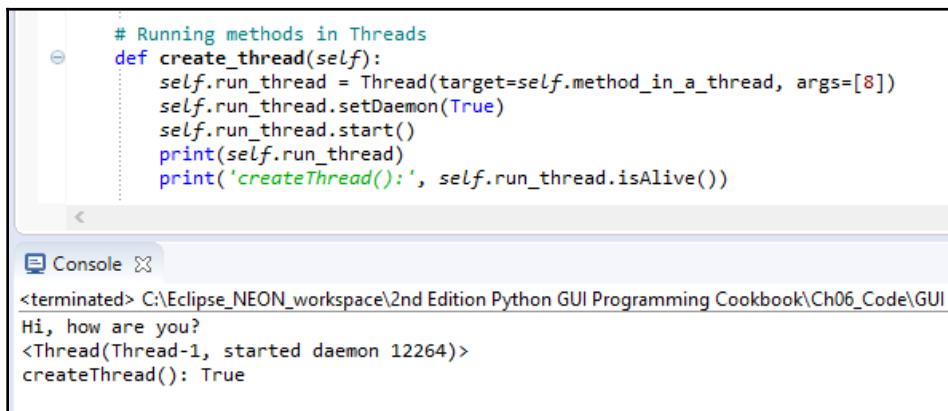
Console Output:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch06_Code\GUI
Hi, how are you?
<Thread(Thread-1, started 11304)>
createThread(): True
Exception in thread Thread-1:
Traceback (most recent call last):
  File "C:\Python36\lib\threading.py", line 916, in _bootstrap_inner
    self.run()
  File "C:\Python36\lib\threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
  File "C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch06
    self.scrol.insert(tk.INSERT, str(idx) + '\n')
  File "C:\Python36\lib\tkinter\__init__.py", line 3266, in insert
    self.tk.call((self._w, 'insert', index, chars) + args)
RuntimeError: main thread is not in main loop
```

5. Add `self.run_thread.setDaemon(True)` to the code:

```
# Running methods in Threads
def create_thread(self):
    self.run_thread = Thread(target=self.method_in_a_thread,
                           args=[8])
    self.run_thread.setDaemon(True)           # <== add this line
    self.run_thread.start()
    print(self.run_thread)
```

6. Run the modified code, click the button, and then close the GUI:



The screenshot shows the Eclipse IDE interface. In the top editor window, there is Python code. In the bottom 'Console' window, the output of running the code is shown.

```
# Running methods in Threads
def create_thread(self):
    self.run_thread = Thread(target=self.method_in_a_thread, args=[8])
    self.run_thread.setDaemon(True)
    self.run_thread.start()
    print(self.run_thread)
    print('createThread():', self.run_thread.isAlive())
```

Console output:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch06_Code\GUI
Hi, how are you?
<Thread(Thread-1, started daemon 12264)>
createThread(): True
```

Let's now see how the recipe works!

How it works...

In the following code, `run_thread` is a local variable, which we only access within the scope of the method inside which we created `run_thread`:

```
# Running methods in Threads
def create_thread(self):
    run_thread = Thread(target=self.method_in_a_thread, args=[8])
    run_thread.start()
```

By turning the local variable into a class instance attribute, we can then check if the thread is still running by calling `isAlive` on it from another method. In `GUI_multiple_threads_stopping_a_thread.py`, we have elevated our local `run_thread` variable to an instance attribute of our class. This enables us to access the `self.run_thread` variable from any method in our class.

When we click the button and then exit the GUI before the thread has finished, we get a runtime error.

Threads are expected to finish their assigned task, so when we close the GUI before the thread has completed, according to the error, Python tells us that the thread we started is not in the main event loop. We can solve this by turning the thread into a *daemon*, which will then execute as a background task. What this gives us is that as soon as we close our GUI, which is our main thread that starts other threads, the daemon threads will cleanly exit. We do this by calling the `setDaemon(True)` method on the thread before we start the thread.

When we now click the button and exit our GUI before the thread has completed its assigned task, we no longer get any errors. While there is a `start` method to make threads run, surprisingly there isn't really an equivalent stop method.

In this recipe, we are running a method in a thread, which prints numbers to our `ScrolledText` widget. When we exit our GUI, we are no longer interested in the thread that used to print to our widget, so by turning the thread into a *daemon*, we can exit our GUI cleanly.

Let's move on to the next recipe.

How to use queues

A Python queue is a data structure that implements the **First In, First Out (FIFO)** paradigm, basically working like a pipe. You shovel something into the pipe on one side and it falls out on the other side of the pipe.

The main difference between this queue shoveling and shoveling mud into physical pipes is that, in Python queues, things do not get mixed up. You put one unit in, and that unit comes back out on the other side. Next, you place another unit in (say, for example, an instance of a class), and this entire unit will come back out on the other end as one piece. It comes back out at the other end in the exact order we inserted code into the queue.



A queue is not a stack in which we push and pop data. A stack is a **Last In, First Out (LIFO)** data structure.

Queues are containers that hold data being fed into the queue from potentially different data sources. We can have different clients providing data to the queue whenever those clients have data available. Whichever client is ready to send data to our queue sends it, and we can then display this data in a widget or send it forward to other modules.

Using multiple threads to complete assigned tasks in a queue is very useful when receiving the final results of processing and displaying them. The data is inserted at one end of the queue and then comes out of the other end in an ordered fashion, FIFO.

Our GUI might have five different button widgets such that each kicks off a different task that we want to display in our GUI in a widget (for example, a `ScrolledText` widget). These five different tasks take a different amount of time to complete.

Whenever a task has completed, we immediately need to know this and display this information in our GUI. By creating a shared Python queue and having the five tasks write their results to this queue, we can display the result of whichever task has been completed immediately using the FIFO approach.

Getting ready

As our GUI is ever-increasing in its functionality and usefulness, it starts to talk to networks, processes, and websites, and will eventually have to wait for data to be made available for the GUI to display.

Creating queues in Python solves the problem of waiting for data to be displayed inside our GUI.

How to do it...

In order to create queues in Python, we have to import the `Queue` class from the `queue` module. Add the following statement toward the top of the GUI module:

1. Open `GUI_multiple_threads_starting_a_thread.py` and save it as `GUI_queues.py`.
2. Make the following changes to the code:

```
from threading import Thread  
from queue import Queue
```

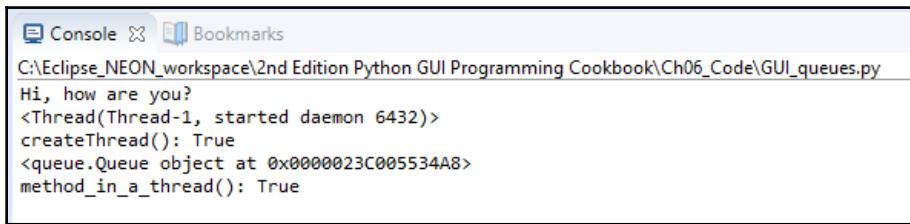
3. Add the following method:

```
def use_queues(self):
    gui_queue = Queue()           # create queue instance
    print(gui_queue)              # print instance
```

4. Modify the `click_me` method:

```
# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get())
    self.create_thread()
    self.use_queues()
```

5. Run the preceding code and observe the output as illustrated in the following screenshot:



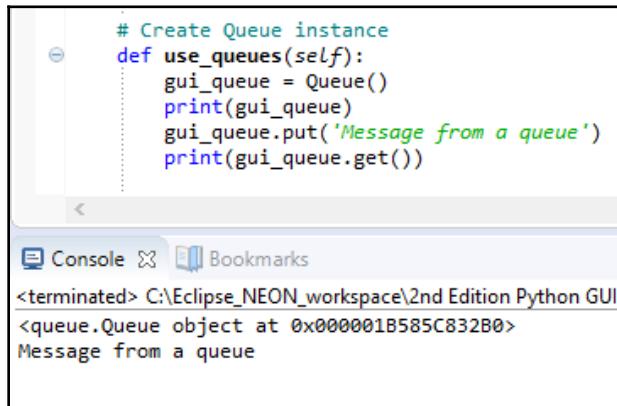
The screenshot shows a terminal window with the title bar "Console Bookmarks". The current directory is "C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch06_Code\GUI_queues.py". The output of the code execution is displayed:

```
Hi, how are you?
<Thread(Thread-1, started daemon 6432)>
createThread(): True
<queue.Queue object at 0x0000023C005534A8>
method_in_a_thread(): True
```

6. Modify `use_queues` to use `put` and `get`:

```
# Create Queue instance
def use_queues(self):
    gui_queue = Queue()
    print(gui_queue)
    gui_queue.put('Message from a queue')
    print(gui_queue.get())
```

7. Run the preceding code and observe the output, as illustrated in the following screenshot:



The screenshot shows the Eclipse IDE interface with a code editor and a console window. The code in the editor is:

```
# Create Queue instance
def use_queues(self):
    gui_queue = Queue()
    print(gui_queue)
    gui_queue.put('Message from a queue')
    print(gui_queue.get())
```

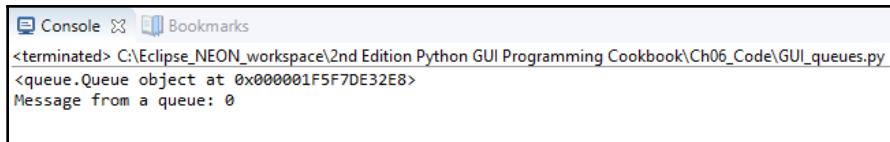
The console window below shows the output:

```
Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI
<queue.Queue object at 0x000001B585C832B0>
Message from a queue
```

8. Write a loop to place many messages into Queue:

```
# Create Queue instance
def use_queues(self):
    gui_queue = Queue()
    print(gui_queue)
    for idx in range(10):
        gui_queue.put('Message from a queue: ' + str(idx))
    print(gui_queue.get())
```

9. Run the preceding code:



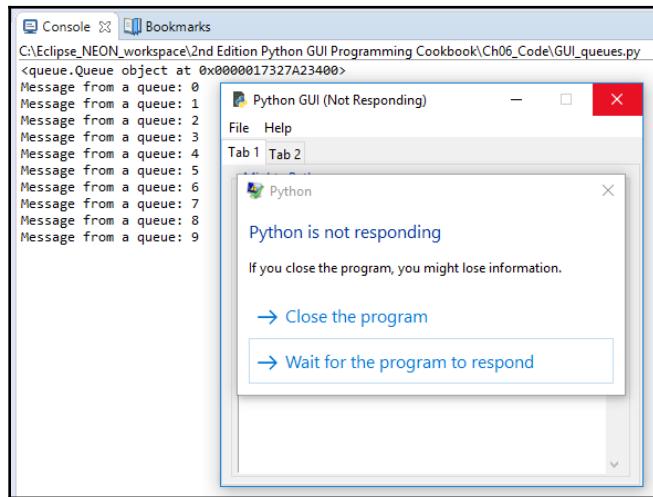
The screenshot shows the Eclipse IDE interface with a code editor and a console window. The console window shows the output of the previous code:

```
Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch06_Code\GUI_queues.py
<queue.Queue object at 0x000001F5F7DE32E8>
Message from a queue: 0
```

10. Add a while loop:

```
# Create Queue instance
def use_queues(self):
    gui_queue = Queue()
    print(gui_queue)
    for idx in range(10):
        gui_queue.put('Message from a queue: ' + str(idx))
    while True:
        print(gui_queue.get())
```

11. Run the preceding code to see the following result:



Now, let's consider the scenario of the endless loop:

1. Open `GUI_queues.py` and save it as `GUI_queues_put_get_loop_endless_threaded.py`.
2. Make the following changes to start `self.run_thread` as a background daemon thread:

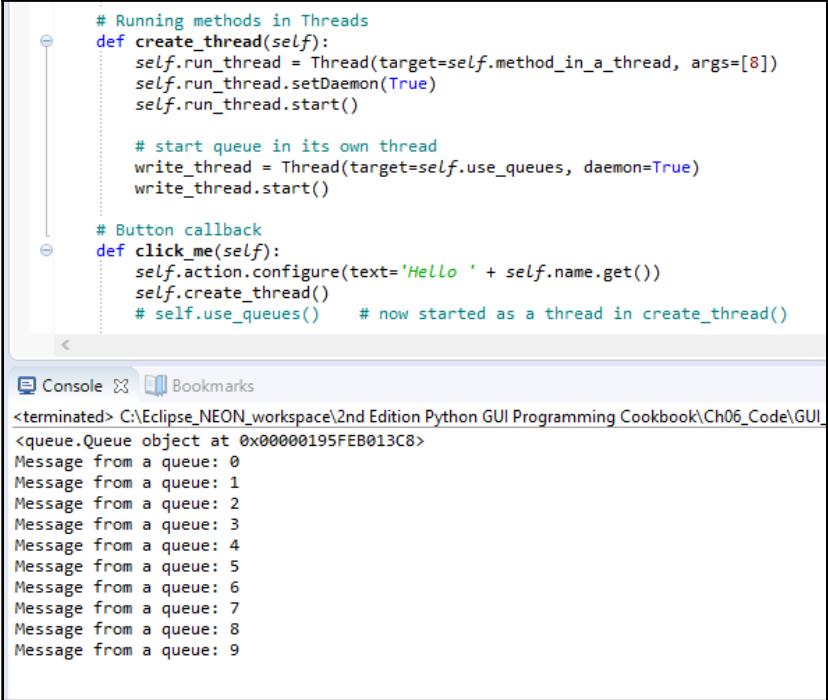
```
# Running methods in Threads
def create_thread(self):
    self.run_thread = Thread(target=self.method_in_a_thread,
                            args=[8])
    self.run_thread.setDaemon(True)
    self.run_thread.start()

# start queue in its own thread
write_thread = Thread(target=self.use_queues, daemon=True)
write_thread.start()
```

3. In the `click_me` method, we comment out `self.use_queues()` and now call `self.create_thread()` instead:

```
# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get())
    self.create_thread()
    # now started as a thread in create_thread()
    # self.use_queues()
```

4. Run the code to see the following result:



The screenshot shows the Eclipse IDE interface. On the left is the code editor with Python code. On the right is the terminal window showing the output of the code execution.

Code Editor Content:

```
# Running methods in Threads
def create_thread(self):
    self.run_thread = Thread(target=self.method_in_a_thread, args=[8])
    self.run_thread.setDaemon(True)
    self.run_thread.start()

    # start queue in its own thread
    write_thread = Thread(target=self.use_queues, daemon=True)
    write_thread.start()

# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get())
    self.create_thread()
    # self.use_queues() # now started as a thread in create_thread()
```

Terminal Output:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch06_Code\GUI_
<queue.Queue object at 0x00000195FE8013C8>
Message from a queue: 0
Message from a queue: 1
Message from a queue: 2
Message from a queue: 3
Message from a queue: 4
Message from a queue: 5
Message from a queue: 6
Message from a queue: 7
Message from a queue: 8
Message from a queue: 9
```

Let's go behind the scenes to understand the code better.

How it works...

In `GUI_queues.py`, we first add `import` statements and then create a new method to create `Queue`. We call the method within our button click event.



In the code, we create a local `Queue` instance that is only accessible within this method. If we wish to access this queue from other places, we have to turn it into an instance attribute of our class by using the `self` keyword, which binds the local variable to the entire class, making it available from any other method within our class. In Python, we often create class instance variables in the `__init__(self)` method, but Python is very pragmatic and enables us to create those attributes anywhere in the code.

Now we have an instance of a queue. We can see that this works by printing it out.

In order to put the data into the queue, we use the `put` command. In order to get the data out of the queue, we use the `get` command.

Running the code results in the message first being placed in `Queue`, then being taken out of `Queue`, and then being printed to the console. We have placed 10 messages into `Queue`, but we are only getting the first one out. The other messages are still inside `Queue`, waiting to be taken out in a FIFO fashion. In order to get all the messages that have been placed into `Queue` out, we can create an endless loop.



While this code works, unfortunately, it freezes our GUI. In order to fix this, we have to call the method in its own thread, as we did in the previous recipes.

We do this in `GUI_queues_put_get_loop_endless_threaded.py`.

When we now click the button, the GUI no longer freezes and the code works. We created `Queue` and placed messages into one side of `Queue` in a FIFO fashion. We got the messages out of `Queue` and then printed them to the console (`stdout`). We realized that we have to call the method in its own thread because, otherwise, our GUI might freeze.

Let's move on to the next recipe.

Passing queues among different modules

In this recipe, we will pass queues around different modules. As our GUI code increases in complexity, we want to separate the GUI components from the business logic, separating them out into different modules. Modularization allows us to reuse code and also makes the code more readable.

Once the data to be displayed in our GUI comes from different data sources, we will face latency issues, which is what queues solve. By passing instances of `Queue` among different Python modules, we are separating the different concerns of the modules' functionalities.



The GUI code ideally would only be concerned with creating and displaying widgets and data.

The business logic modules' job is only to do the business logic and supply the resulting data to the GUI.

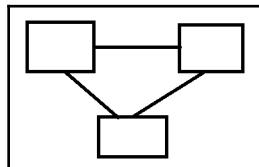
We have to combine the two elements, ideally using as few relationships among the different modules as possible, reducing code interdependence.



The coding principle of avoiding unnecessary dependencies is usually called *loose coupling*. This is a very important principle and I strongly encourage you to look into it, understand it, and apply it to your own coding projects.

In order to understand the significance of *loose coupling*, we can draw some boxes on a whiteboard or a piece of paper. One box represents our GUI class and code, while the other boxes represent business logic, databases, and so on.

Next, we draw lines between the boxes, graphing out the interdependencies between those boxes, which are our Python modules, as shown here:



While these three boxes connected via three lines might look a little simple, this is what you really would draw on a whiteboard in a software team meeting. I have left out any labels, but one box could be labeled **UI**, another **database**, and a third **business processing logic**.



The fewer lines we have between our Python boxes, the more *loosely coupled* our design is.

Getting ready

In the previous recipe, *How to use queues*, we started to use queues. In this recipe, we will pass instances of `Queue` from our main GUI thread to other Python modules, which will enable us to write to the `ScrolledText` widget from another module while keeping our GUI responsive.

How to do it...

1. First, we create a new Python module in our project. Let's call it `Queues.py`. We'll place a function into it (no OOP necessary yet). Sequentially, we can state it as follows:
 1. Create a new Python module and name it `Queues.py`.
 2. Write the following code into this module to place messages into the instance queue:

```
def write_to_scrol(inst):
    print('hi from Queue', inst)
    for idx in range(10):
        inst.gui_queue.put('Message from a queue: ' +
str(idx))
        inst.create_thread(6)
```

2. The next steps show how we shall import this newly created module:
 1. Open `GUI_queues_put_get_loop_endless_threaded.py` and save it as `GUI_passing_queues_member.py`.
 2. Make the following changes to invoke the function from the module we are importing:

```
import Ch06_Code.Queues as bq      # bq; background queue

class OOP():
    # Button callback
    def click_me(self):
        # Passing in the current class instance (self)
        print(self)
        bq.write_to_scrol(self)
```

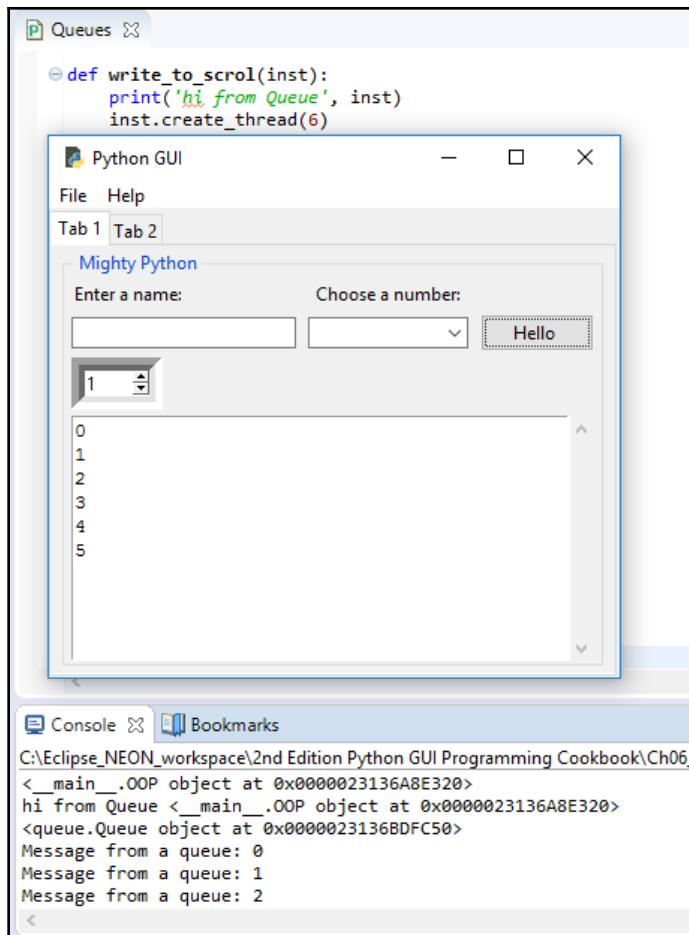
3. In `GUI_passing_queues_member.py`, create an instance of Queue:

```
class OOP():
    def __init__(self):
        # Create a Queue
        self.gui_queue = Queue()
```

4. Modify the `use_queues` method:

```
def use_queues(self):
    # Now using a class instance member Queue
    while True:
        print(self.gui_queue.get())
```

5. Running the code yields the following result:



Let's go behind the scenes to understand the code better.

How it works...

First, we create a new Python module, `Queues.py`. The `write_to_scrol` function within it accepts an instance of a class. We use this instance to access the methods and attributes of the class.



Here, we are relying on the knowledge that our class instance has the two methods we are accessing within the function.

In `GUI_passing_queues_member.py`, we first import the `Queues` module, alias it to `bq`, and then we use it to call the function residing in the `Queues` module.



Aliasing the module to `bq` is probably not the best name. I meant it to mean *background queue* because it runs threads as daemons in the background. I am not changing the alias in this third edition as I have used it in the first two editions of this book, for reasons of consistency.

In the `click_me` button callback method, we are passing `self` into this function. This enables us to use all of the GUI methods from another Python module.

The imported module contains the `write_to_scrol` function we are calling:

```
def write_to_scrol(inst):
    print('hi from Queue', inst)
    inst.create_thread(6)
```

By passing in a `self`-reference from the class instance to the function that the class is calling in another module, we now have access to all our GUI elements from other Python modules.

`gui_queue` is an instance attribute and `create_thread` is a method, and both are defined in `GUI_passing_queues_member.py`, and we are accessing them via the passed-in `self`-reference inside the `Queues` module.

We create `Queue` as an instance attribute of our class, placing a reference to it in the `__init__` method of the `GUI_passing_queues_member.py` class.

Now we can put messages into the queue from our new module by simply using the passed-in class reference to our GUI. Notice `inst.gui_queue.put` in the `Queues.py` code:

```
def write_to_scrol(inst):
    print('hi from Queue', inst)
    for idx in range(10):
        inst.gui_queue.put('Message from a queue: ' + str(idx))
    inst.create_thread(6)
```

After we modified the `use_queues` method, the `create_thread` method in our GUI code only reads from the `Queue`, which got filled in by the business logic residing in our new module, which has separated the logic from our GUI module.

In order to separate the GUI widgets from the functionality that expresses the business logic, we created a class, made a queue an instance attribute of this class, and, by passing an instance of the class into a function residing in a different Python module, we now have access to all the GUI widgets, as well as the queue.



This is the magic of OOP. In the middle of a class, we pass ourselves into a function we are calling from within the class using the `self` keyword.

This recipe is an example of when it makes sense to program in OOP.

Let's move on to the next recipe.

Using dialog widgets to copy files to your network

This recipe shows us how to copy files from your local hard drive to a network location. We will do this by using one of Python's `tkinter` built-in dialogs, which enables us to browse our hard drive. We can then select a file to be copied.

This recipe also shows us how to make `Entry` widgets read-only and to default `Entry` to a specified location, which speeds up the browsing of our hard drive.

Getting ready

We will extend **Tab 2** of the GUI we were building in the previous recipe, *Passing queues among different modules*.

How to do it...

Add the following code to the GUI in the `create_widgets()` method toward the bottom, where we created **Tab Control 2**. The parent of the new widget frame is `tab2`, which we created at the very beginning of the `create_widgets()` method. As long as you place the following code physically under the creation of `tab2`, it will work:

1. Open `GUI_passing_queues_member.py` and save it as `GUI_copy_files.py`.
2. Make the following changes:

```
#####
def create_widgets(self):
    # Create Tab Control
    tabControl = ttk.Notebook(self.win)
    # Add a second tab
```

```
tab2 = ttk.Frame(tabControl)
# Make second tab visible
tabControl.add(tab2, text='Tab 2')

# Create Manage Files Frame
mngFilesFrame = ttk.LabelFrame(tab2, text=' Manage Files: ')
mngFilesFrame.grid(column=0, row=1, sticky='WE', padx=10, pady=5)

# Button Callback
def getFileName():
    print('hello from getFileName')

# Add Widgets to Manage Files Frame
lb = ttk.Button(mngFilesFrame, text="Browse to File...", command=getFileName)
lb.grid(column=0, row=0, sticky=tk.W)

file = tk.StringVar()
self.entryLen = scrol_w
self.fileEntry = ttk.Entry(mngFilesFrame, width=self.entryLen, textvariable=file)
self.fileEntry.grid(column=1, row=0, sticky=tk.W)

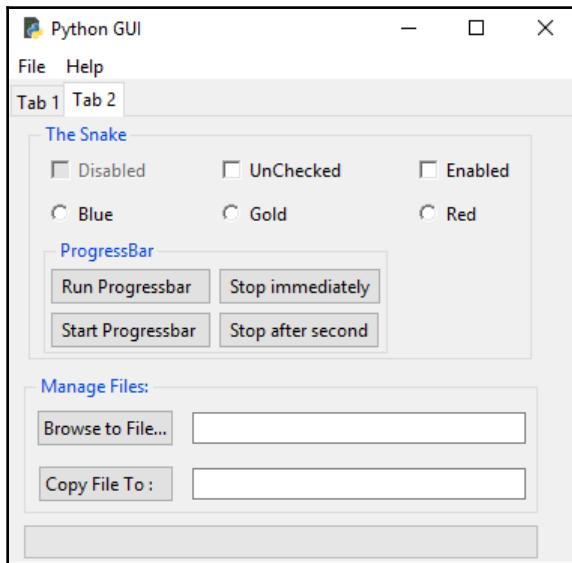
logDir = tk.StringVar()
self.netwEntry = ttk.Entry(mngFilesFrame, width=self.entryLen, textvariable=logDir)
self.netwEntry.grid(column=1, row=1, sticky=tk.W)

def copyFile():
    import shutil
    src = self.fileEntry.get()
    file = src.split('/')[-1]
    dst = self.netwEntry.get() + '/' + file
    try:
        shutil.copy(src, dst)
        msg.showinfo('Copy File to Network', 'Success: File copied.')
    except FileNotFoundError as err:
        msg.showerror('Copy File to Network', '*** Failed to copy file! ***\n\n' + str(err))
    except Exception as ex:
        msg.showerror('Copy File to Network', '*** Failed to copy file! ***\n\n' + str(ex))

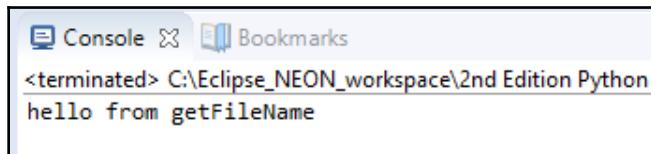
cb = ttk.Button(mngFilesFrame, text="Copy File To : ", command=copyFile)
cb.grid(column=0, row=1, sticky=tk.E)
```

```
# Add some space around each label
for child in mngFilesFrame.winfo_children():
    child.grid_configure(padx=6, pady=6)
```

3. Running the code creates the following GUI:



4. Click the **Browse to File...** button:



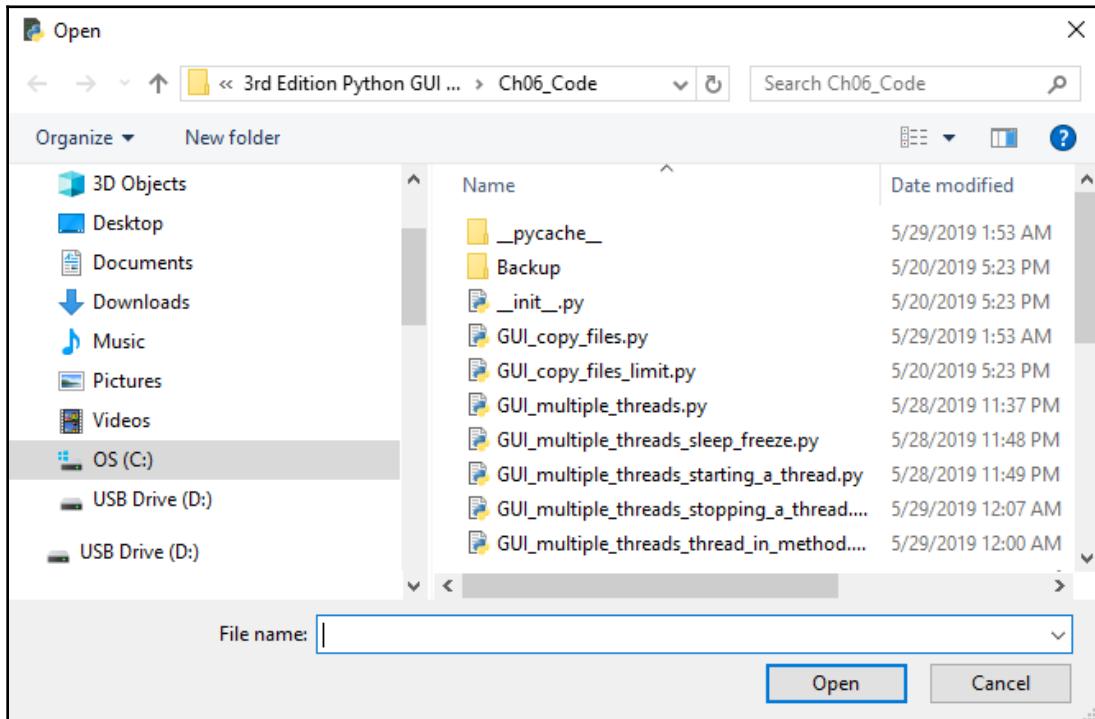
1. Open `GUI_copy_files.py`.
2. Add the following two import statements:

```
from tkinter import filedialog as fd
from os import path
```

5. Create the following function:

```
def getFileName():
    print('hello from getFileName')
    fDir = path.dirname(__file__)
    fName = fd.askopenfilename(parent=self.win, initialdir=fDir)
```

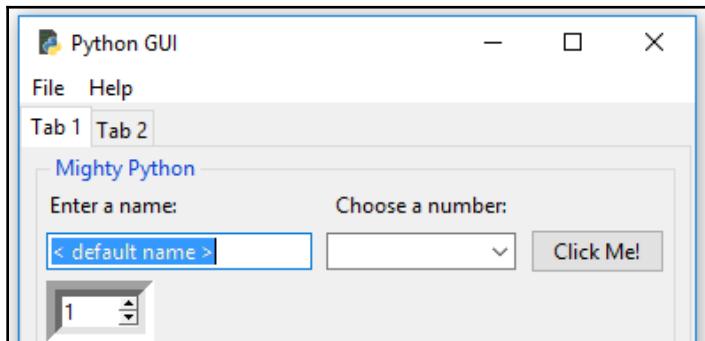
6. Run the code and click the **Browse To** button:



7. Add the following two lines of code to the creation of the Entry widget:

```
# Adding a Textbox Entry widget
self.name = tk.StringVar()
self.name_entered = ttk.Entry(mighty, width=24,
textvariable=self.name)
self.name_entered.grid(column=0, row=1, sticky='W')
self.name_entered.delete(0, tk.END)
self.name_entered.insert(0, '< default name >')
```

8. Run the code and see the following result:



9. Now, open `GUI_copy_files.py` and add the following code:

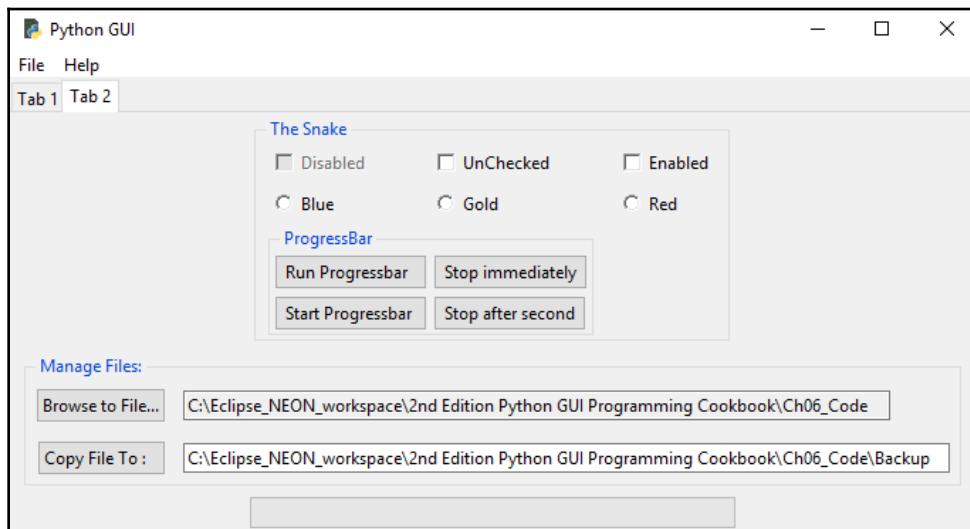
```
# Module level GLOBALS
GLOBAL_CONST = 42
fDir      = path.dirname(__file__)
netDir = fDir + 'Backup'

def __init__(self):
    self.createWidgets()
    self.defaultFileEntries()

def defaultFileEntries(self):
    self.fileEntry.delete(0, tk.END)
    self.fileEntry.insert(0, fDir)
    if len(fDir) > self.entryLen:
        self.fileEntry.config(width=len(fDir) + 3)
        self.fileEntry.config(state='readonly')

    self.netwEntry.delete(0, tk.END)
    self.netwEntry.insert(0, netDir)
    if len(netDir) > self.entryLen:
        self.netwEntry.config(width=len(netDir) + 3)
```

10. Running `GUI_copy_files.py` results in the following screenshot:



11. Open `GUI_copy_files.py` and add the following code:

```
# Module level GLOBALS
GLOBAL_CONST = 42

from os import makedirs
fDir = path.dirname(__file__)
netDir = fDir + 'Backup'
if not path.exists(netDir):
    makedirs(netDir, exist_ok = True)
```

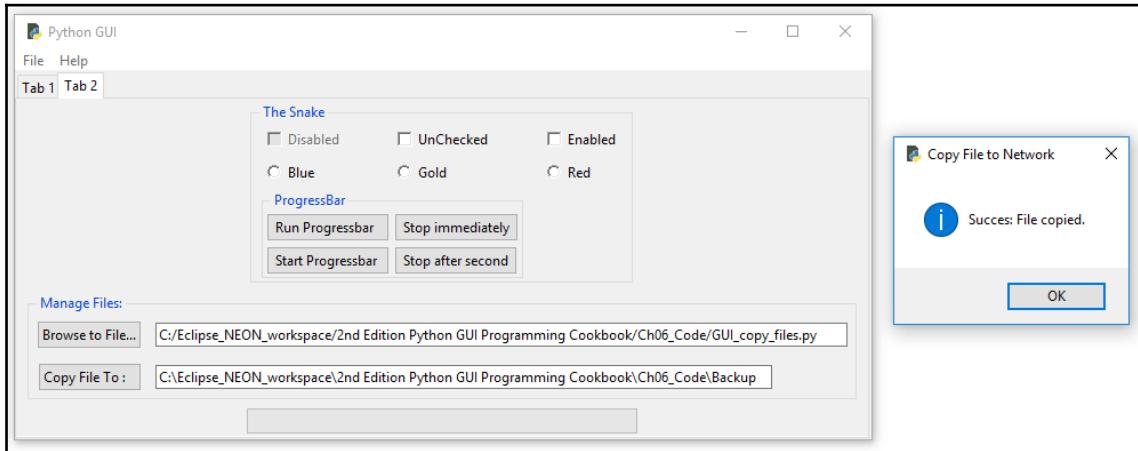
Once we click the button that invokes the `copyFile()` function, we import the required module.

12. Open `GUI_copy_files.py` and add the following code:

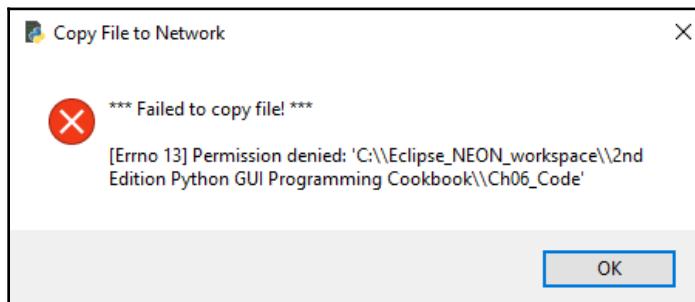
```
from tkinter import messagebox as msg
def copyFile():
    import shutil                      #import module within function
    src = self.fileEntry.get()
    file = src.split('/')[-1]
    dst = self.netwEntry.get() + '+' + file
    try:
        shutil.copy(src, dst)
        msg.showinfo('Copy File to Network', 'Success: File copied.')
```

```
except FileNotFoundError as err:  
    msg.showerror('Copy File to Network',  
                 '*** Failed to copy file! ***\n\n' + str(err))  
except Exception as ex:  
    msg.showerror('Copy File to Network',  
                 '*** Failed to copy file! ***\n\n' + str(ex))
```

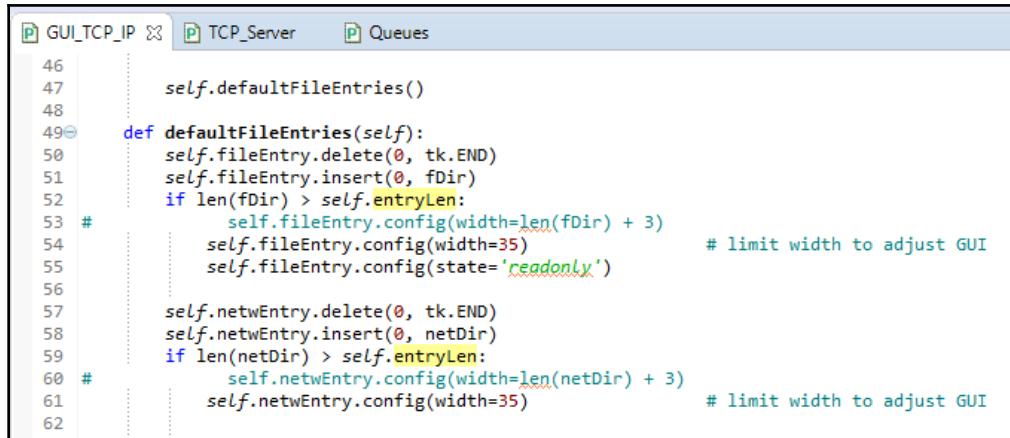
13. Run the code, browse to a file, and click the **Copy** button:



14. Run the code, but don't browse and click the **Copy** button:



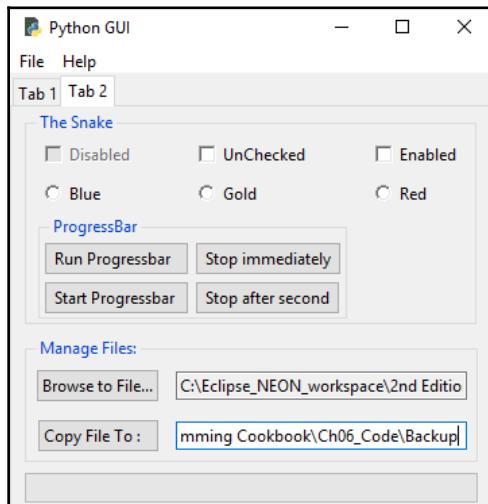
15. Open `GUI_copy_files.py` and save it as `GUI_copy_files_limit.py`.
16. Add the following code:



The screenshot shows a code editor window with tabs for "GUI_TCP_IP", "TCP_Server", and "Queues". The "TCP_Server" tab is active, displaying the following Python code:

```
46
47     self.defaultFileEntries()
48
49 def defaultFileEntries(self):
50     self.fileEntry.delete(0, tk.END)
51     self.fileEntry.insert(0, fDir)
52     if len(fDir) > self.entryLen:
53         self.fileEntry.config(width=len(fDir) + 3)
54         self.fileEntry.config(width=35) # limit width to adjust GUI
55         self.fileEntry.config(state='readonly')
56
57     self.netwEntry.delete(0, tk.END)
58     self.netwEntry.insert(0, netDir)
59     if len(netDir) > self.entryLen:
60         self.netwEntry.config(width=len(netDir) + 3)
61         self.netwEntry.config(width=35) # limit width to adjust GUI
62
```

17. Run the preceding code to observe the output, as illustrated in the following screenshot:



Let's go behind the scenes to understand the code better.

How it works...

In `GUI_copy_files.py`, we add two buttons and two entries to **Tab 2** of our GUI. We are not yet implementing the functionality of our button callback function.

Clicking the **Browse to File...** button currently prints `hello` from `getFileName` to the console. We can use the `tkinter` built-in file dialogs after adding the `import` statements.

We can now use the dialogs in our code. Instead of hardcoding a path, we can use Python's `os` module to find the full path to where our GUI module resides. Clicking the **Browse to File...** button now opens up the `askopenfilename` dialog. We can now open a file in this directory or browse to a different directory. After selecting a file and clicking the **Open** button in the dialog, we will save the full path to the file in the `fName` local variable.

It would be nice if, when we opened our Python `askopenfilename` dialog widget, we would automatically default to a directory so that we would not have to browse all the way to where we were looking for a particular file to be opened. It is best to demonstrate how to do this by going back to our GUI **Tab 1**, which is what we will do next.

We can default the values into `Entry` widgets. Back on our **Tab 1**, this is very easy. When we now run the GUI, the `name_entered` entry has a default value. We can get the full path to the module we are using, and then we can create a new subfolder just below it. We can do this as a module-level global variable, or we can create the subfolder within a method.

We set the defaults for both the `Entry` widgets and, after setting them, we make the local file `Entry` widget read-only.



This order is important. We have to first populate the entry before we make it read-only.

We are also selecting **Tab 2** before calling the main event loop and no longer set the focus into the `Entry` of **Tab 1**. Calling `select` on our notebook of `tkinter` is zero-based, so by passing in the value of 1, we select **Tab 2**:

```
# Place cursor into name Entry
# name_entered.focus()                      # commented out
tabControl.select(1)                         # displayTab 2 at GUI startup
```

As we are not all on the same network, this recipe uses the local hard drive in place of a network.

A UNC path is a **Universal Naming Convention**, and what this means is that by using double backslashes instead of the typical C:\, we can access a server on a network.



You just have to use the UNC and replace C:\ with \\<servername>\<folder>.

This example can be used to back up our code to a backup directory, which we can create if it does not exist by using `os.makedirs`. After selecting a file to copy to somewhere else, we import the Python `shutil` module. We need the full path to the source of the file to be copied and a network or local directory path, and then we append the filename to the path where we will copy it using `shutil.copy`.



`shutil` is shorthand notation for shell utility.

We also give feedback to the user via a message box to indicate whether the copying succeeded or failed. In order to do this, we import `messagebox` and alias it to `msg`.

In the next code, we mix two different approaches of where to place our `import` statements. In Python, we have some flexibility that other languages do not provide. We typically place all of the `import` statements toward the very top of each of our Python modules so that it is clear which modules we are importing. At the same time, a modern coding approach is to place the creation of variables close to the function or method where they are first being used.

In the code, we import the message box at the top of our Python module, but then we also import the `shutil` Python module in a function. Why would we wish to do this? Does this even work? The answer is yes, it does work, and we are placing this `import` statement into a function because this is the only place in our code where we actually do need this module.

If we never call this method, then we will never import the module this method requires. In a sense, you can view this technique as the **lazy initialization design pattern**. If we don't need it, we don't import it until we really do require it in our Python code. The idea here is that our entire code might require, let's say, 20 different modules. At runtime, which modules are really needed depends upon the user interaction. If we never call the `copyFile()` function, then there is no need to import `shutil`.

When we now run our GUI, browse to a file, and click **Copy**, the file is copied to the location we specified in our `Entry` widget.

If the file does not exist or we forgot to browse to a file and are trying to copy the entire parent folder, the code will let us know this as well because we are using Python's built-in exception handling capabilities.

Our new `Entry` widgets did expand the width of the GUI. While it is sometimes nice to be able to see the entire path, at the same time, it pushes other widgets, making our GUI look not so good. We can solve this by restricting the width parameter of our `Entry` widgets. We do this in `GUI_copy_files_limit.py`. This results in a limited GUI size. We can right-arrow in the enabled `Entry` widget to get to the end of this widget.

We are copying files from our local hard drive to a network by using the Python shell utility. As most of us are not connected to the same local area network, we simulate the copying by backing up our code to a different local folder.

We are using one of `tkinter` dialog controls, and by defaulting the directory paths, we can increase our efficiency in copying files.

Let's move on to the next recipe.

Using TCP/IP to communicate via networks

This recipe shows you how to use `sockets` to communicate via *TCP/IP*. In order to achieve this, we need both an *IP address* and a *port number*.

In order to keep things simple and independent of changing internet IP addresses, we will create our own local TCP/IP server and client, and we will learn how to connect the client to the server and read data via a TCP/IP connection.

We will integrate this networking capability into our GUI by using the queues we created in the previous recipes.



TCP/IP short for **Transmission Control Protocol/Internet Protocol**, which is a set of networking protocols that allows two or more computers to communicate.

Getting ready

We will create a new Python module, which will be the TCP server.

How to do it...

One way to implement a TCP server in Python is to inherit from the `socketserver` module. We subclass `BaseRequestHandler` and then override the inherited `handle` method. In very few lines of Python code, we can implement a TCP server:

1. Create a new Python module and save it as `TCP_Server.py`.
2. Add the following code to create the TCP server and a `start` function:

```
from socketserver import BaseRequestHandler, TCPServer

class RequestHandler(BaseRequestHandler):
    # override base class handle method
    def handle(self):
        print('Server connected to: ', self.client_address)
        while True:
            rsp = self.request.recv(512)
            if not rsp: break
            self.request.send(b'Server received: ' + rsp)

    def start_server():
        server = TCPServer(('', 24000), RequestHandler)
        server.serve_forever()
```

3. Open `Queues.py` and add the following code to create a socket and use it:

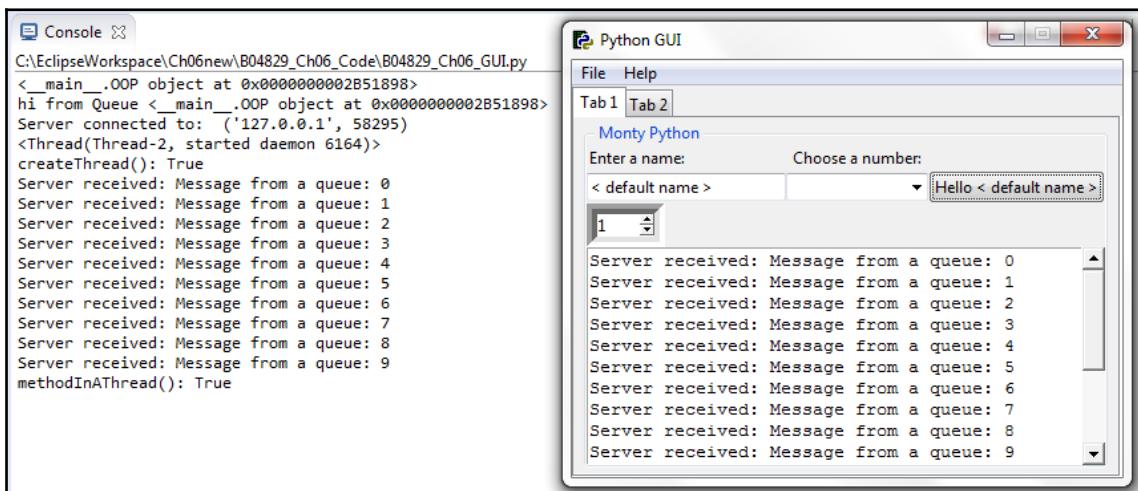
```
# using TCP/IP
from socket import socket, AF_INET, SOCK_STREAM

def write_to_scrol_TCP(inst):
    print('hi from Queue', inst)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.connect(('localhost', 24000))
    for idx in range(10):
        sock.send(b'Message from a queue: ' + bytes(str(idx).encode()))
    recv = sock.recv(8192).decode()
    inst.gui_queue.put(recv)
    inst.create_thread(6)
```

4. Open `GUI_copy_files_limit.py` and save it as `GUI_TCP_IP.py`.
5. Add the following code to start the TCP server in its own thread:

```
class OOP():
    def __init__(self):
        # Start TCP/IP server in its own thread
        svrT = Thread(target=start_server, daemon=True)
        svrT.start()
```

6. Run the code and click the **Click Me!** button on **Tab 1**:



Let's go behind the scenes to understand the code better.

How it works...

In `TCP_Server.py`, we are passing our `RequestHandler` class into a `TCPServer` initializer. The empty single quotes are a shortcut for `localhost`, which is our own PC. This is the IP address of `127.0.0.1`. The second item in the tuple is the *port number*. We can choose any port number that is not in use on our local PC.



We have to make sure that we are using the same port on the client side of the TCP connection; otherwise, we would not be able to connect to the server.

Of course, we have to start the server first before clients can connect to it. We will modify our `Queues.py` module to become the TCP client. When we now click the **Click Me!** button, we are calling `bq.write_to_scrol_TCP(self)`, which then creates the socket and connection.

This is all the code we need to talk to the TCP server. In this example, we are simply sending some bytes to the server and the server sends them back, prepending some strings before returning the response.

This shows the principle of how TCP communications via networks work.



Once we know how to connect to a remote server via TCP/IP, we will use whatever commands are designed by the protocol of the program we are interested in communicating with. The first step is to connect before we can send commands to specific applications residing on a server.

In the `write_to_scrol_TCP` function, we use the same loop as before, but now we will send the messages to the TCP server. The server modifies the received message and then sends it back to us. Next, we place it into the GUI class instance queue, which, as in the previous recipes, runs in its own thread:

```
sock.send(b'Message from a queue: ' + bytes(str(idx).encode())) )
```

Note the `b` character before the string and then the rest of the required casting.

We start the TCP server in its own thread in the initializer of the OOP class.



In Python 3, we send strings over sockets in binary format. Adding the integer index now becomes a little bit convoluted as we have to cast it to a string, encode it, and then cast the encoded string into bytes!

Clicking the **Click Me!** button on **Tab 1** now creates the output in our `ScrolledText` widget as well as on the console, and the response because of the use of threads is very fast. We created a TCP server to simulate connecting to a server in our local area network or on the internet. We turned our `queues` module into a TCP client. We are running both the queue and the server in their own background thread, which keeps our GUI very responsive.

Let's move on to the next recipe.

Using urlopen to read data from websites

This recipe shows how we can easily read entire web pages by using some of Python's built-in modules. We will display the web page data first in its raw format and then decode it, and then we will display it in our GUI.

Getting ready

We will read the data from a web page and then display it in the `ScrolledText` widget of our GUI.

How to do it...

First, we create a new Python module and name it `URL.py`. We then import the required functionality to read web pages using Python. We can do this in very few lines of code:

1. Create a new module and name it `URL.py`.
2. Add the following code to open and read the URL:

```
from urllib.request import urlopen
link = 'http://python.org/'
try:
    http_rsp = urlopen(link)
    print(http_rsp)
    html = http_rsp.read()
    print(html)
    html_decoded = html.decode()
    print(html_decoded)
except Exception as ex:
    print('*** Failed to get Html! ***\n\n' + str(ex))
else:
    return html_decoded
```

3. Run the preceding code and observe the following output:

```
<terminated> GUI_URL.py [C:\Python37\python.exe]
hi from Queue <__main__.OOP object at 0x000002154CB06F98>
Server connected to: ('127.0.0.1', 53684)
Server received: Message from a queue: 0
<http.client.HTTPResponse object at 0x000002154F5C5E48>
b'<!doctype html><!--[if lt IE 7]>  <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">
<!doctype html>
<!--[if lt IE 7]>  <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">  <![endif]-->
<!--[if IE 7]>    <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->
<!--[if IE 8]>    <html class="no-js ie8 lt-ie9">          <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr">  <!--<![endif]-->

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

4. Compare the result to the official Python web page we just read:

Welcome to Python.org

Python Software Foundation [US] | https://www.python.org

Python PSF Docs PyPI Jobs Community

python™

Python 3: List comprehensions

```
>>> fruits = ['Banana', 'Apple', 'Lime']
>>> loud_fruits = [fruit.upper() for fruit in fruits]
>>> print(loud_fruits)
['BANANA', 'APPLE', 'LIME']

# List and the enumerate function
>>> list(enumerate(fruits))
[(0, 'Banana'), (1, 'Apple'), (2, 'Lime')]
```

Compound Data Types

Lists (known as arrays in other languages) are one of the compound data types that Python understands. Lists can be indexed, sliced and manipulated with other built-in functions.

[More about lists in Python 3](#)

1 2 3 4 5

Let's consider the next scenario:

1. Open URL.py.
2. Place the code into a function:

```
from urllib.request import urlopen
link = 'http://python.org/'

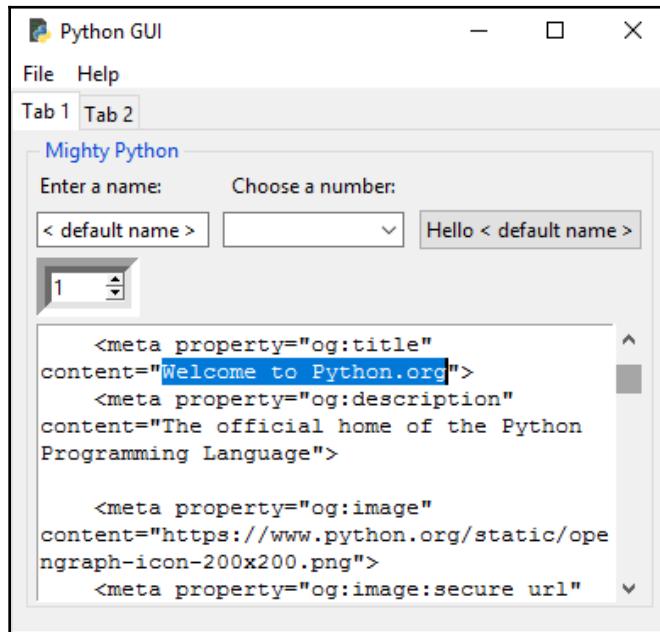
def get_html():
    try:
        http_rsp = urlopen(link)
        print(http_rsp)
        html = http_rsp.read()
        print(html)
        html_decoded = html.decode()
        print(html_decoded)
    except Exception as ex:
        print('*** Failed to get Html! ***\n\n' + str(ex))
    else:
        return html_decoded
```

3. Open GUI_TCP_IP.py from the previous recipe and save it as GUI_URL.py.
4. Import the URL module and modify the click_me method:

```
import Ch06_Code.URL as url

# Button callback
def click_me(self):
    self.action.configure(text='Hello ' + self.name.get())
    bq.write_to_scrol(self)
    sleep(2)
    html_data = url.get_html()
    print(html_data)
    self.scrol.insert(tk.INSERT, html_data)
```

5. Run the code, the output of which is as follows:



The next section talks about the process in detail.

How it works...

We wrap the `URL.py` code in a `try...except` block similar to Java and C#. This is a modern approach to coding, which Python supports. Whenever we have code that might not complete, we can experiment with this code and, if it works, all is fine. If the block of code in the `try...except` block does not work, the Python interpreter will throw one of several possible exceptions, which we can then catch. Once we have caught the exception, we can decide what to do next.

There is a hierarchy of exceptions in Python, and we can also create our own classes that inherit from and extend the Python exception classes. In the following code, we are mainly concerned that the URL we are trying to open might not be available, so we wrap our code within a `try...except` code block. If the code succeeds in opening the requested URL, all is fine. If it fails, maybe because our internet connection is down, we fall into the exception part of the code and print out that an exception has occurred.



You can read more about Python exception handling at <https://docs.python.org/3.7/library/exceptions.html>.

By calling `urlopen` on the official Python website, we get the entire data as one long string. The first `print` statement prints this long string out to the console. We then call `decode` on the result, and this time we get a little over 1,000 lines of web data, including some white space. We also print out `type` for calling `urlopen`, which is an `http.client.HTTPResponse` object. Actually, we print it out first.

Next, we display this data in our GUI inside the `ScrolledText` widget. In order to do so, we have to connect our new module, which reads the data from the web page to our GUI. In order to do this, we need a reference to our GUI, and one way to do this is by tying our new module to the **Tab 1** button callback. We can return the decoded HTML data from the Python web page to the `Button` widget, which we can then place into the `ScrolledText` control.

We turn our `URL.py` code into a function and return the data to the calling code. We can now write the data from our button callback method to the `ScrolledText` control by first importing the new module and then inserting the data into the widget. We also give it some sleep after the call to `write_to_scrol`.

In `GUI_URL.py`, the HTML data is now displayed in our GUI widget.

7

Storing Data in Our MySQL Database via Our GUI

In this chapter, we will learn how to install and use a MySQL database and connect it to our GUI.

MySQL is a full-fledged **Structured Query Language (SQL)** database server and comes with a very nice GUI of its own so that we can view and work with the data. We will create a database, insert data into our database, and then see how we can modify, read, and delete data.

Data storage in a SQL database is essential for software programs written in Python. All of our data currently only exists in memory and we want to make it persistent so that we do not lose our data once we close our running Python program.

Here, you will learn how to increase your programming skills by adding SQL to your programming toolbox.



The first recipe in this chapter will show you how to install the free MySQL Community Edition.

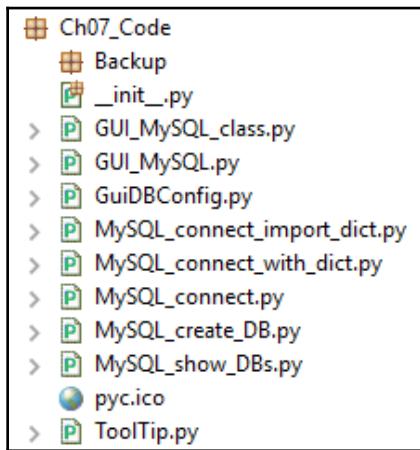
After successfully connecting to a running instance of our MySQL server, we will design and create a database that will accept a book title, which could be our own journal or a quote we found somewhere on the internet. We will require a page number for the book, which could be blank (`NULL` in SQL terms), and then we will insert the quote we like from a book, journal, website, or a friend into our MySQL database using our GUI, which we built using Python 3.7 or later.

We will insert, modify, delete, and display our favorite quotes using our Python GUI to issue these SQL commands and to display the data.



CRUD is a database term you may have come across before that is an abbreviation for the four basic SQL commands, that is, **Create**, **Read**, **Update**, and **Delete**.

Here is an overview of the Python modules for this chapter:



In this chapter, we will enhance our Python GUI by connecting the GUI to a MySQL database. We will cover the following recipes:

- Installing and connecting to a MySQL server from Python
- Configuring the MySQL database connection
- Designing the Python GUI database
- Using the SQL INSERT command
- Using the SQL UPDATE command
- Using the SQL DELETE command
- Storing and retrieving data from our MySQL database
- Using MySQL Workbench

Installing and connecting to a MySQL server from Python

Before we can connect to a MySQL database, we have to connect to the *MySQL server*. In order to do this, we need to know the IP address of the MySQL server as well as the port it is listening on.

We also have to be a registered user with a password in order to be *authenticated* by the MySQL server.

Getting ready

You will need to have access to a running MySQL server instance, as well as have administrator privileges in order to create databases and tables.

How to do it...

Let's look at how to install and connect to a MySQL server from Python:

1. Download the MySQL Installer.



There is a free MySQL Community Edition available from the official MySQL website. You can download and install it on your local PC from <http://dev.mysql.com/downloads/windows/installer/>.

2. Run the installation:

Choosing the right file:

- If you have an online connection while running the MySQL Installer, choose the [mysql-installer-web-community](#) file.
- If you do NOT have an online connection while running the MySQL Installer, choose the [mysql-installer-community](#) file.

Note: MySQL Installer is 32 bit, but will install both 32 bit and 64 bit binaries.

Online Documentation

- [MySQL Installer Documentation](#) and [Change History](#)

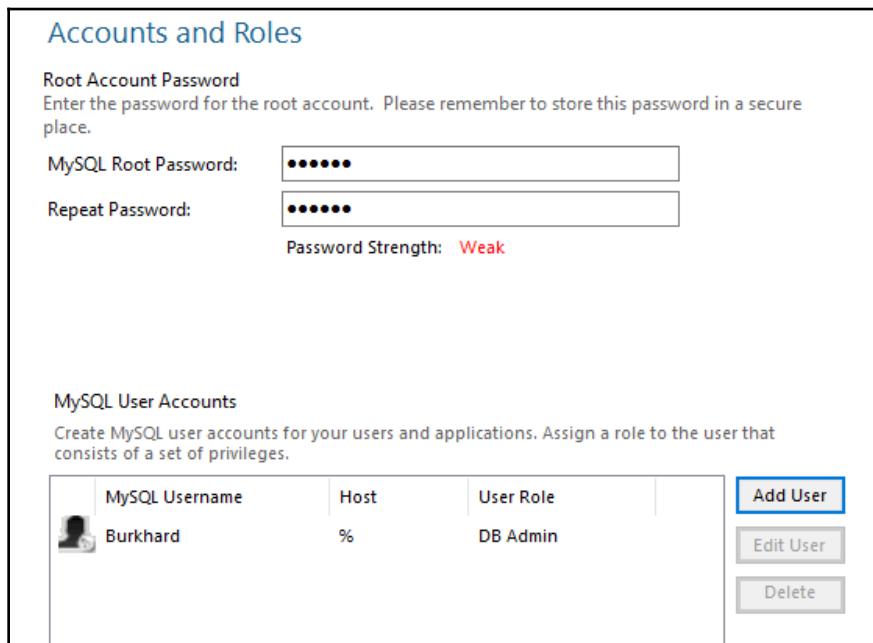
Please report any bugs or inconsistencies you observe to our [Bugs Database](#).

Thank you for your support!

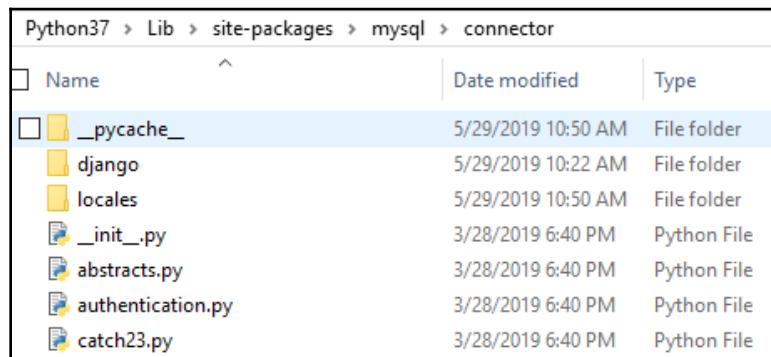
The screenshot shows the MySQL Installer download page. At the top, there's a header with the MySQL logo and a search bar. Below it, a section titled "Generally Available (GA) Releases" is highlighted in orange. Underneath, the title "MySQL Installer 8.0.16" is displayed. A dropdown menu for "Select Operating System" is open, showing "Microsoft Windows" as the selected option. To the right, a link says "Looking for previous GA versions?". The main content area lists two download options for Windows (x86, 32-bit) MSI Installers:

File Type	Version	File Size	Action
Windows (x86, 32-bit), MSI Installer (mysql-installer-web-community-8.0.16.0.msi)	8.0.16	20.0M	Download
Windows (x86, 32-bit), MSI Installer	8.0.16	373.4M	Download

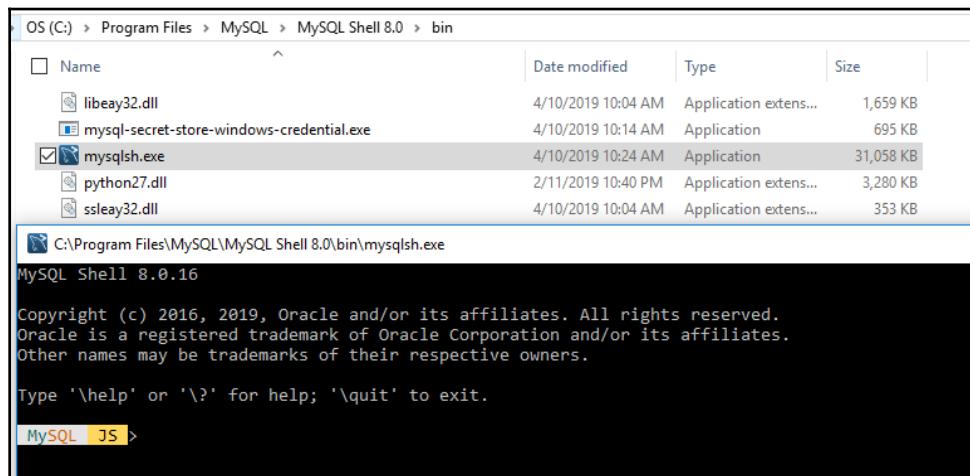
3. Choose a password for the `root` user and, optionally, add more users:



4. Verify that you have the `\Python37\Lib\site-packages\mysql\connector` folder:



5. Open the mysqlsh.exe executable and double-click on it to run it:



6. Type \sql in the prompt to get into SQL mode.
7. In the MySQL> prompt, type SHOW DATABASES. Then, press *Enter*:

A screenshot of a terminal window titled C:\Program Files\MySQL\MySQL Shell 8.0\bin\mysqlsh.exe. The MySQL shell is running in SQL mode. The user connects to the root user at localhost. After entering the password, the shell displays the MySQL connection id (21), server version (8.0.16), and the fact that no default schema is selected. The user then attempts to run the SHOW DATABASES command, which fails with a SyntaxError: Unexpected identifier. The user then switches to SQL mode using the \sql command. Finally, the user runs the SHOW DATABASES command again, which successfully lists the databases: information_schema, mysql, performance_schema, sakila, sys, and world. The output shows 6 rows in set (0.0009 sec).

```
MySQL> \connect --mc root@localhost
Creating a Classic session to 'root@localhost'
Please provide the password for 'root@localhost': *****
Save password for 'root@localhost'? [Y]es/[N]o/Ne[v]er (default No): Y
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 21
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

MySQL> localhost:3306 ssl> SHOW DATABASES;
SyntaxError: Unexpected identifier

MySQL> localhost:3306 ssl> \sql
Switching to SQL mode... Commands end with ;

MySQL> localhost:3306 ssl> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sakila         |
| sys            |
| world          |
+-----+
6 rows in set (0.0009 sec)
```

8. Create a new Python module and save it as MySQL_connect.py:

```
import mysql
conn = mysql.connector.connect(user=<adminUser>,
                                password=<adminPwd>, host='127.0.0.1')
print(conn)
conn.close()
```

9. If running the preceding code results in the following output, then we have successfully connected:

Console X

```
<terminated> MySQL_connect_import_dict.py [C:\Python37\python.exe]
<mysql.connector.connection_cext.CMySQLConnection object at 0x000002114F92A390>
```

Let's go behind the scenes to understand the code better.

How it works...

First, we downloaded and then installed the MySQL version that matches our operating system.



During the installation process, you will choose a password for the `root` user, and you can also add more users. I recommend that you add yourself as a **DB Admin** and choose a password as well.

In this chapter, we are using the latest MySQL Community Server release, that is, 8.0.16.



SQL stands for **Structured Query Language** and is sometimes pronounced **sequel**. It uses a **Set** mathematical approach, which is based on mathematics and **set theory**. You can find out more at https://en.wikipedia.org/wiki/Set_theory.

In order to connect to MySQL, we may need to install a special Python connector driver. This driver will allow us to talk to the MySQL server from Python. There is a freely available driver on the MySQL website (<http://dev.mysql.com/doc/connector-python/en/index.html>) and it comes with a very nice online tutorial.



When I did a brand new installation of the latest version of MySQL, the Python connector was automatically installed. Therefore, you may not have to install it after all. It is good to know, though, just in case you run into any issues and need to install it yourself.

One way to verify that we have installed the correct driver and that it lets Python talk to MySQL is by looking into the Python `site-packages` directory. If your `site-packages` directory has a new `MySQL` folder that contains a `connector` subfolder, the installation was successful. We did this in *step 4*.

In *step 5*, we verified that our MySQL server installation actually worked by using the MySQL Shell.



Your path might be different, especially if you are on macOS or Linux: `<path to>\Program Files\MySQL\MySQL Shell 8.0\bin`.

Next, we verified that we can achieve the same results using Python 3.7.



Replace the placeholder bracketed names, that is, `<adminUser>` and `<adminPwd>`, with the real credentials you are using in your MySQL installation.

We have to be able to connect to the MySQL server. By default, we are in JavaScript `JS` mode. We can change that by typing `\sql` in the prompt to get into `SQL` mode. Now, we can use `SQL` commands. We did this in *steps 6* and *7*.

If you are unable to connect to the MySQL server via the *Command Shell* or the Python `mysqlclient`, then something probably went wrong during the installation. If this is the case, try uninstalling MySQL, rebooting your PC, and then running the installation again.

In order to connect our GUI to a MySQL server, we need to be able to connect to the server with administrative privileges. We also need to do this if we want to create our own database. If the database already exists, then we just need the authorization rights to connect, insert, update, and delete data. We will create a new database on a MySQL server in the next recipe.

Configuring the MySQL database connection

In the previous recipe, we used the shortest way to connect to a MySQL server, that is, by hardcoding the credentials that are required for authentication in the `connect` method. While this is a fast approach for early development, we definitely do not want to expose our MySQL server credentials to anyone. Instead, we want to *grant* permission to specific users so that they can access databases, tables, views, and related database commands.

A much safer way to be authenticated by a MySQL server is by storing the credentials in a configuration file, which is what we will do in this recipe. We will use our configuration file to connect to the MySQL server and then create our own database on the MySQL server.

We will use this database in all of the recipes in this chapter.



Getting ready

Access to a running MySQL server with administrator privileges is required to run the code shown in this recipe.



The previous recipe shows how to install the free *Community Edition* of MySQL server. The administrator privileges will allow you to implement this recipe.

How to do it...

Let's look at how to perform this recipe:

1. First, we will create a dictionary in the same module where the `MySQL_connect.py` code is. Sequentially, we will do the following:
 1. Open `MySQL_connect.py` and save it as `MySQL_connect_with_dict.py`.

2. Add the following code to the module:

```
# create dictionary to hold connection info
dbConfig = {
    'user': <adminName>,          # use your admin name
    'password': <adminPwd>,       # use your real password
    'host': '127.0.0.1',          # IP address of localhost
}
```

2. Write the following code below dbConfig:

```
import mysql.connector
# unpack dictionary credentials
conn = mysql.connector.connect(**dbConfig)
print(conn)
```

3. Run the code to make sure it works.

4. Create a new module, GuiDBConfig.py, and place the following code in it:

```
# create dictionary to hold connection info
dbConfig = {
    'user': <adminUser>,          # your user name
    'password': <adminPwd>,       # your password
    'host': '127.0.0.1',          # IP address
}
```

5. Now, open MySQL_connect_with_dict.py and save it as MySQL_connect_import_dict.py.

6. Import GuiDBConfig and unpack the dictionary, as shown here:

```
import GuiDBConfig as guiConf
# unpack dictionary credentials
conn = mysql.connector.connect(**guiConf.dbConfig)
print(conn)
```

7. Create a new Python module and save it as MySQL_create_DB.py. Next, add the following code:

```
import mysql.connector
import Ch07_Code.GuiDBConfig as guiConf

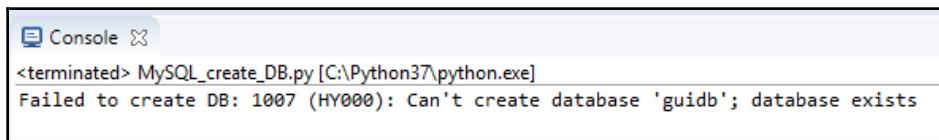
GUIDB = 'GuiDB'

# unpack dictionary credentials
conn = mysql.connector.connect(**guiConf.dbConfig)

cursor = conn.cursor()
```

```
try:  
    cursor.execute("CREATE DATABASE {}  
                    DEFAULT CHARACTER SET 'utf8'".format(GUIDB))  
except mysql.connector.Error as err:  
    print("Failed to create DB: {}".format(err))  
  
conn.close()
```

8. Execute MySQL_create_DB.py twice:

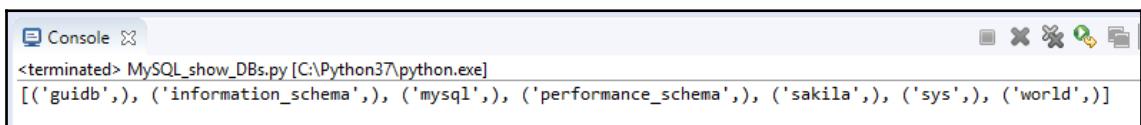


```
Console  
<terminated> MySQL_create_DB.py [C:\Python37\python.exe]  
Failed to create DB: 1007 (HY000): Can't create database 'guidb'; database exists
```

9. Create a new Python module and save it as MySQL_show_DBs.py. Then, add the following code:

```
import mysql.connector  
import GuiDBConfig as guiConf  
  
# unpack dictionary credentials  
conn = mysql.connector.connect(**guiConf.dbConfig)  
  
cursor = conn.cursor()  
  
cursor.execute("SHOW DATABASES")  
print(cursor.fetchall())  
  
conn.close()
```

10. Running the preceding code gives us the following output:



```
Console  
<terminated> MySQL_show_DBs.py [C:\Python37\python.exe]  
[('guidb',), ('information_schema',), ('mysql',), ('performance_schema',), ('sakila',), ('sys',), ('world',)]
```

Let's go behind the scenes to understand the code better.

How it works...

First, we created a dictionary and saved our connection credentials in the Python dictionary.

Next, in the `connect` method, we unpacked the dictionary values. Take a look at the following code:

```
mysql.connector.connect('user': <adminName>, 'password': <adminPwd>,
'host': '127.0.0.1')
```

Instead of using this code, we use `(**dbConfig)`, which achieves the same thing but is shorter.

This results in the same successful connection to the MySQL server, but the difference is that the connection method no longer exposes any mission-critical information.



A database server is critical to your mission. You will realize this once you have lost your valuable data and can't find any recent backup!

Please note that placing the same username, password, database, and so on into a dictionary in the same Python module does not eliminate the risk of having the credentials seen by anyone perusing the code.

In order to increase database security, we had to move the dictionary into its own Python module. We called the new Python module `GuiDBConfig.py`.

We then imported this module and unpacked the credentials, as we did previously.



Once we placed this module into a secure place, separated from the rest of the code, we achieved a better level of security for our MySQL data.

Now that we know how to connect to MySQL and have administrator privileges, we could create our own database by issuing SQL commands.

In order to execute commands to MySQL, we created a cursor object from the connection object.

A cursor is usually a pointer to a specific row in a database table that we can move up or down the table, but here, we used it to create the database itself. We wrapped the Python code into a `try...except` block and used the built-in error codes of MySQL to tell us if anything went wrong.

We can verify that this block works by executing the database-creating code twice. The first time, it will create a new database in MySQL, and the second time, it will print out an error message stating that this database already exists.

We can verify which databases exist by executing the `SHOW DATABASES` command using the very same cursor object syntax. Instead of issuing the `CREATE DATABASE` command, we create a cursor and use it to execute the `SHOW DATABASES` command, the result of which we fetch and print to the console output.



We retrieve the results by calling the `fetchall` method on the cursor object.

Running the `MySQL_show_DBs.py` code shows us which databases currently exist in our MySQL server instance. As we saw from the output, MySQL ships with several built-in databases, such as `information_schema`. We successfully created our own `guidb` database, which is shown in the output. All of the other databases that were illustrated come shipped with MySQL.

Note how, even though we specified the database when we created it in mixed-case letters as `GuiDB`, the `SHOW DATABASES` command shows all the existing databases in MySQL in lowercase and displays our database as `guidb`.



The physical MySQL files are stored on the hard drive according to the `my.ini` file, which, on a Windows 10 installation, may be located at `C:\ProgramData\MySQL\MySQL Server 8.0`. Within this `.ini` file, you can find the following configuration path to the `Data` folder:

```
# Path to the database root  
datadir=C:/ProgramData/MySQL/MySQL Server 8.0/Data
```

Let's move on to the next recipe.

Designing the Python GUI database

Before we start creating tables and inserting data into them, we have to design the database. Unlike changing local Python variable names, changing a database schema once it has been created and loaded with data is not that easy.

We would have to `DROP` the table, which means we would lose all the data that was in the table. So, before dropping a table, we would have to extract the data, save the data in a temporary table or other data format, and then `DROP` the table, recreate it, and finally reimport the original data.

I hope you are getting the picture of how tedious this could be.

Designing our GUI MySQL database means that we need to think about what we want our Python application to do with it and then choose names for our tables that match the intended purpose.

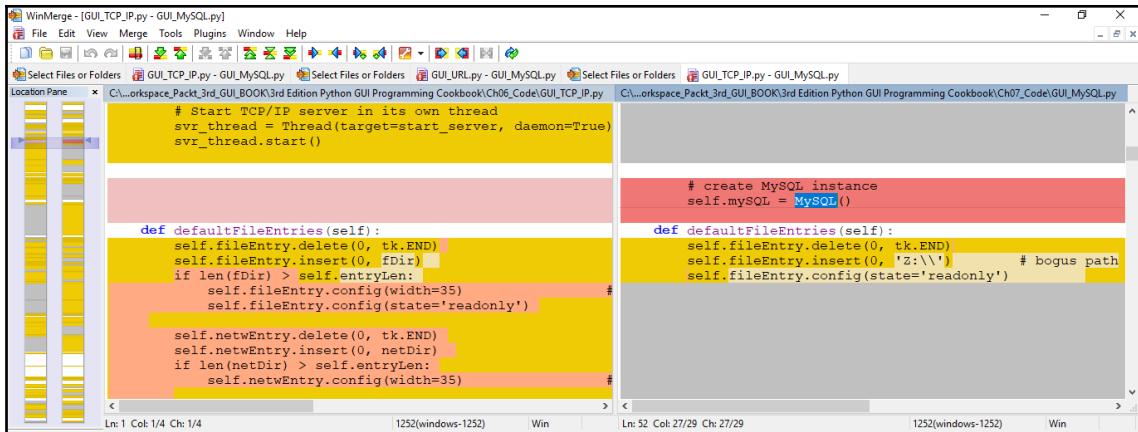
Getting ready

We will be working with the MySQL database we created in the previous recipe, *Configuring the MySQL database connection*. A running instance of MySQL is necessary and the two previous recipes show you how to install MySQL, all the necessary additional drivers, and how to create the database we are using in this chapter.

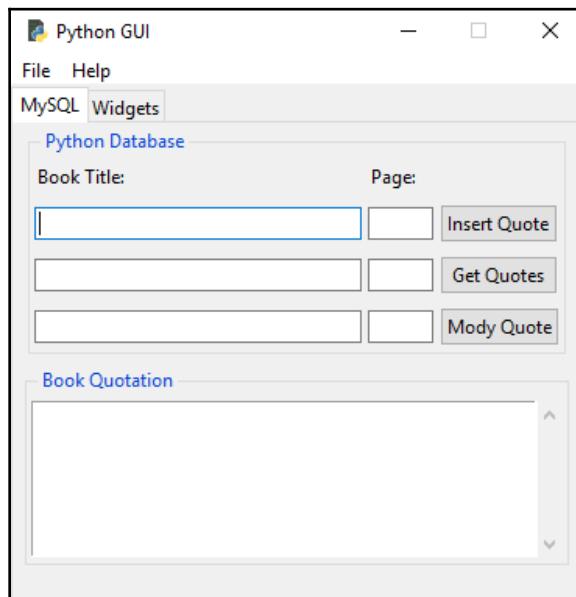
How to do it...

In this recipe, we are starting with the `GUI_TCP_IP.py` file from the previous chapter. We will move the widgets from our Python GUI between the two tabs we created in the previous recipes in order to organize our Python GUI so that it can connect to a MySQL database. Let's take a look at how we can complete this recipe:

1. Open `GUI_TCP_IP.py` and save it as `GUI_SQL.py`.
2. Download the full code from the Packt website.
3. Use a tool such as WinMerge to compare the two versions of the GUI:



4. Run the code located in `GUI_SQL.py`. You will observe the following output:



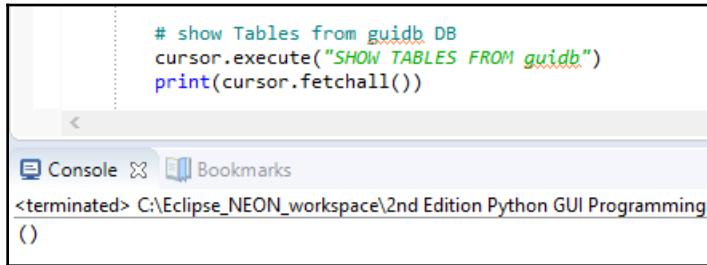
1. Now, open `MySQL_create_DB.py` and save it as `MySQL_show_DB.py`.
2. Replace the `try...catch` block with the following code:

```
# unpack dictionary credentials
conn = mysql.connect(**guiConf.dbConfig)
# create cursor
cursor = conn.cursor()
```

```
# execute command
cursor.execute("SHOW TABLES FROM guidb")
print(cursor.fetchall())

# close connection to MySQL
conn.close()
```

5. Run the code and observe the output:



The screenshot shows a Python script being run in an Eclipse IDE's Console view. The script contains a single line of code: `cursor.execute("SHOW TABLES FROM guidb")`. The output of the script is displayed below the code, showing the result of the SQL query: `<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming ()`.

1. Create a module similar to `GUI_SQL_class.py`.
2. Add and run the following code:

```
# connect by unpacking dictionary credentials
conn = mysql.connect(**guiConf.dbConfig)

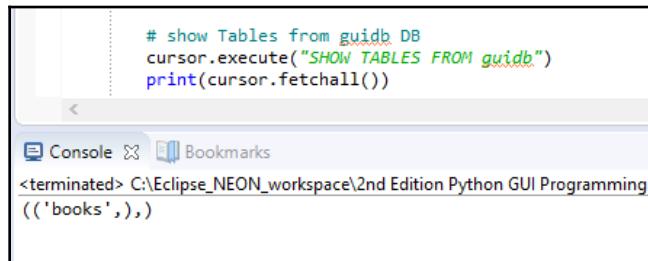
# create cursor
cursor = conn.cursor()

# select DB
cursor.execute("USE guidb")

# create Table inside DB
cursor.execute("CREATE TABLE Books (
    Book_ID INT NOT NULL AUTO_INCREMENT,
    Book_Title VARCHAR(25) NOT NULL,
    Book_Page INT NOT NULL,
    PRIMARY KEY (Book_ID)
) ENGINE=InnoDB")

# close connection to MySQL
conn.close()
```

6. Run the following code, which is located in `GUI_MySQL_class.py`:



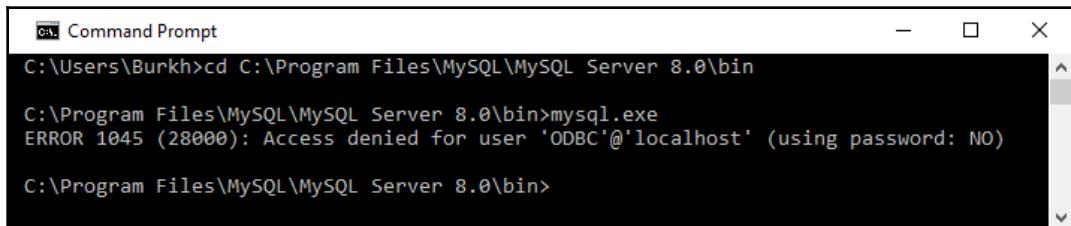
A screenshot of an Eclipse IDE's Python console window. The code in the editor pane is:

```
# show Tables from guidb DB
cursor.execute("SHOW TABLES FROM guidb")
print(cursor.fetchall())
```

The console output pane shows the results of the executed command:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming
(('books',),)
```

7. Open Command Prompt and navigate to `mysql.exe`:



A screenshot of a Windows Command Prompt window titled "Command Prompt". The command entered was:

```
C:\Users\Burkh>cd C:\Program Files\MySQL\MySQL Server 8.0\bin
```

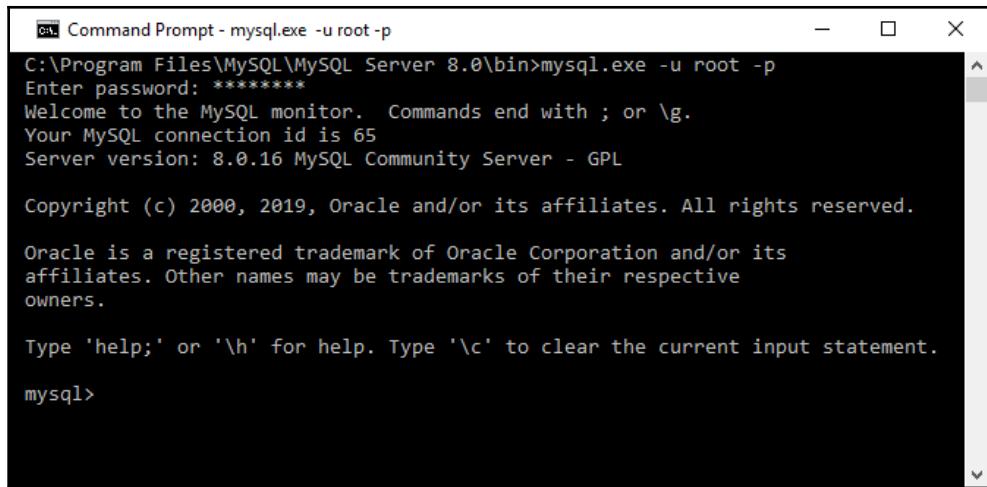
The response shows an error message:

```
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql.exe
ERROR 1045 (28000): Access denied for user 'ODBC'@'localhost' (using password: NO)
```

Followed by the prompt:

```
C:\Program Files\MySQL\MySQL Server 8.0\bin>
```

8. Run `mysql.exe`:



A screenshot of a Windows Command Prompt window titled "Command Prompt - mysql.exe - u root - p". The command entered was:

```
C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql.exe -u root -p
```

The password was entered as "*****". The MySQL monitor starts up with the following welcome message:

```
Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 65
Server version: 8.0.16 MySQL Community Server - GPL

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
```

The prompt "mysql>" is visible at the bottom.

9. Enter the SHOW COLUMNS FROM books; command:

```
mysql> USE guidb
Database changed
mysql> SHOW COLUMNS FROM books;
+-----+-----+-----+-----+
| Field | Type  | Null | Key  | Default | Extra       |
+-----+-----+-----+-----+
| Book_ID | int(11) | NO   | PRI  | NULL    | auto_increment |
| Book_Title | varchar(25) | NO   |       | NULL    |             |
| Book_Page | int(11) | NO   |       | NULL    |             |
+-----+-----+-----+-----+
3 rows in set (0.21 sec)
```

10. Create a second table by running the following code:

```
# select DB
cursor.execute("USE guidb")

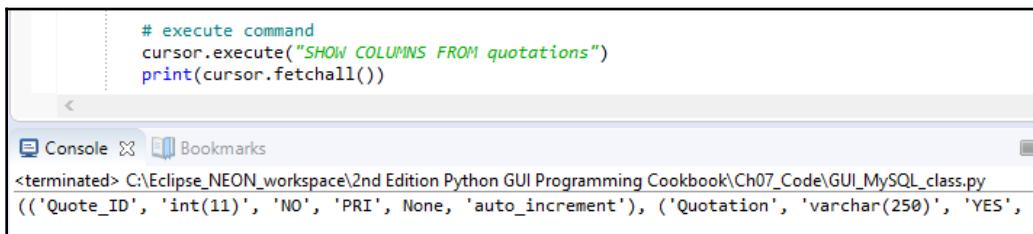
# create second Table inside DB
cursor.execute("CREATE TABLE Quotations (
    Quote_ID INT,
    Quotation VARCHAR(250),
    Books_Book_ID INT,
    FOREIGN KEY (Books_Book_ID)
    REFERENCES Books(Book_ID)
    ON DELETE CASCADE
) ENGINE=InnoDB")
```

11. Execute the SHOW TABLES command:

```
# show Tables from guidb DB
cursor.execute("SHOW TABLES FROM guidb")
print(cursor.fetchall())

Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming
(('books',), ('quotations',))
```

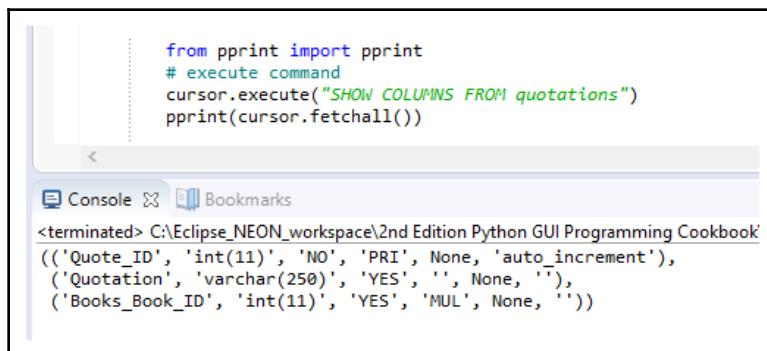
12. Execute the SHOW COLUMNS command:



```
# execute command
cursor.execute("SHOW COLUMNS FROM quotations")
print(cursor.fetchall())
```

Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch07_Code\GUI_MySQL_class.py
([('Quote_ID', 'int(11)', 'NO', 'PRI', None, 'auto_increment'), ('Quotation', 'varchar(250)', 'YES', None), ('Books_Book_ID', 'int(11)', 'YES', 'MUL', None, '')])

13. Execute SHOW COLUMNS again with pprint:



```
from pprint import pprint
# execute command
cursor.execute("SHOW COLUMNS FROM quotations")
pprint(cursor.fetchall())
```

Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook
([('Quote_ID', 'int(11)', 'NO', 'PRI', None, 'auto_increment'), ('Quotation', 'varchar(250)', 'YES', None, ''), ('Books_Book_ID', 'int(11)', 'YES', 'MUL', None, '')])

Let's go behind the scenes to understand the code better.

How it works...

We started with the `GUI_TCP_IP.py` file from the previous chapter and reorganized the widgets.

We renamed several widgets and separated the code that accesses the MySQL data to what used to be named **Tab 1**, and we moved the unrelated widgets to what we named **Tab 2** in the previous recipes. We also adjusted some internal Python variable names so that we can understand our code better.

Code readability is a coding virtue and not a waste of time.

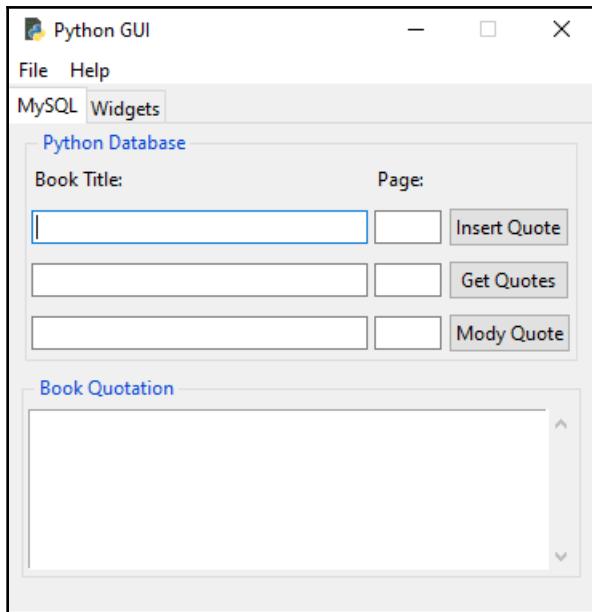


The refactored module is close to 400 lines of Python code and it would take too many pages to show the entire code here. On Windows, we can use a tool called **WinMerge** to compare different Python code modules. I am sure there are similar tools for macOS and Linux.



WinMerge is a great tool for comparing different Python (and other) code modules on Windows. We can use it to look at the differences between code modules. You can download it for free from <https://sourceforge.net/projects/winmerge/>.

Our refactored Python GUI now looks as follows:



We renamed the first tab **MySQL** and created two `LabelFrame` widgets of `tkinter`. We labeled the one on the top `Python Database`, which contains two labels and six `tkinter Entry` widgets, as well as three buttons, which we aligned in four rows and three columns using the `tkinter grid` layout manager. We will enter book titles and pages into the `Entry` widgets. Clicking the buttons will result in either inserting, retrieving, or modifying book quotations. The `LabelFrame` widget at the bottom has a label of **Book Quotation** and the `ScrolledText` widget that is part of this frame will display our books and quotations.

Then, we created two SQL tables to hold our data. The first will hold the data for the book title and book page, which will then join with the second table, which will hold the book quote. We will link the two tables together via *primary key to foreign key relationships*.

So, let's create the first database table now. Before we do that, let's verify that our database does, indeed, have no tables. According to the online MySQL documentation, the command to view the tables that exist in a database is as follows:

```
13.7.6.37 SHOW TABLES Syntax  
SHOW [FULL] TABLES [{FROM | IN} db_name]  
[LIKE 'pattern' | WHERE expr]
```

It is important to note that, in the preceding syntax, arguments in square brackets, such as FULL, are optional, while arguments in curly braces, such as FROM, are required for the SHOW TABLES command. The pipe symbol between FROM and IN means that the MySQL syntax requires one or the other.

When we execute the SQL command in `MySQL_show_DB.py`, we get the expected result, which is an empty tuple showing us that our database currently has no tables.

We can also select the database by executing the `USE <DB>` command. By doing this, we don't have to pass it into the `SHOW TABLES` command because we have already selected the database we want to talk to.



All the SQL code is located in `GUI_MySQL_class.py` and we import this into `GUI_MySQL.py`.

Now that we know how to verify that our database has no tables, we create some. After creating two tables, we verify that they have truly made it into our database by using the same commands as before.

Doing this, we created the first table, named `Books`.

We can verify that the table has been created in our database by executing the `cursor.execute("SHOW TABLES FROM guidb")` command.

The result is no longer an empty tuple but a tuple that contains a tuple, showing the `books` table we just created.

We can use the MySQL command-line client to view the columns in our table. In order to do this, we have to log in as the `root` user. We also have to append a **semicolon** to the end of the command.



On Windows, you simply double-click the MySQL command-line client shortcut, which is automatically installed during the MySQL installation.

If you don't have a shortcut on your desktop, you can find the executable at the following path for a typical default installation:

```
C:\Program Files\MySQL\MySQL Server 8.0\bin\mysql.exe
```

Without a shortcut to run the MySQL client, you have to pass it some parameters:

- C:\Program Files\MySQL\MySQL Server 8.0\bin\mysql.exe
- -u root
- -p



If double-clicking creates an error, make sure you use the `-u` and `-p` options.

Either double-clicking the shortcut or using the command line with the full path to the executable and passing in the required parameters will bring up the MySQL command-line client, which prompts you to enter the password for the root user.

If you remember the password you assigned to the root user during the installation, you can then run the `SHOW COLUMNS FROM books;` command. This will display the columns of our `books` table from our `guidb` database.



When executing commands in the MySQL client, the syntax is not Pythonic, as it requires a trailing semicolon to complete the statement.

Next, we created the second table, which will store the book and journal quotations. We created it by writing similar code to what we used to create the first table. We verified that we now have two tables by running the same SQL command.

We can see the columns by executing the SQL command using Python:

```
cursor.execute("SHOW COLUMNS FROM quotations")
```

Using the MySQL client might present the data in a better format than Command Prompt. We can also use Python's pretty print (`pprint`) feature for this.

The MySQL client still shows our columns in a clearer format, which can be seen when you run this client.

We designed our Python GUI database and refactored our GUI in preparation to use our new database. Then, we created a MySQL database and created two tables within it.

We verified that the tables made it into our database by using both Python and the MySQL client that ships with the MySQL server.

In the next recipe, we will insert data into our tables.

Using the SQL INSERT command

This recipe presents the entire Python code that shows you how to create and drop MySQL databases and tables, as well as how to display the existing databases, tables, columns, and data of our MySQL instance.

After creating the database and tables, we will insert data into the two tables we will create in this recipe.



We are using a *primary key* to *foreign key* relationship to connect the data of the two tables.

We will go into detail about how this works in the following two recipes, where we will modify and delete the data in our MySQL database.

Getting ready

This recipe builds on the MySQL database we created in the previous recipe, *Designing the Python GUI database*, and also shows you how to drop and recreate the GuiDB.



Dropping the database, of course, deletes all the data the database has in its tables, so we'll show you how to reinsert that data as well.

How to do it...

The entire code in the `GUI_SQL_class.py` module is present in the code folder for this chapter, which you can download from <https://github.com/PacktPublishing/Python-GUI-Programming-Cookbook-Third-Edition>. Let's go through these steps sequentially:

1. Download the code for this chapter.
2. Open `GUI_SQL_class.py` and look at the class methods:

```
import mysql.connector
import Ch07_Code.GuiDBConfig as guiConf

class MySQL():
    # class variable
    GUIDB = 'GuidB'
    #-----
    def connect(self):
        # connect by unpacking dictionary credentials
        # create cursor
    #-----
    def close(self, cursor, conn):
        # close cursor
    #-----
    def showDBs(self):
        # connect to MySQL
    #-----
    def createGuidB(self):
        # connect to MySQL
    #-----
    def dropGuidB(self):
        # connect to MySQL
    #-----
    def useGuidB(self, cursor):
        '''Expects open connection.'''
        # select DB
    #-----
    def createTables(self):
        # connect to MySQL
        # create Table inside DB
    #-----
    def dropTables(self):
        # connect to MySQL
    #-----
    def showTables(self):
        # connect to MySQL
    #-----
    def insertBooks(self, title, page, bookQuote):
```

```
# connect to MySQL
# insert data
#-----
def insertBooksExample(self):
    # connect to MySQL
    # insert hard-coded data
#-----
def showBooks(self):
    # connect to MySQL
#-----
def showColumns(self):
    # connect to MySQL
#-----
def showData(self):
    # connect to MySQL
#-----
if __name__ == '__main__':
    # Create class instance
    mySQL = MySQL()
```

3. Running the preceding code (including the full implementation of the code) creates the following tables and data in the database we created.

4. Open Command Prompt and execute the two SELECT * statements:

```
mysql> USE guidb
Database changed
mysql> SELECT * FROM books;
+-----+-----+-----+
| Book_ID | Book_Title      | Book_Page |
+-----+-----+-----+
|      1 | Design Patterns   |      7 |
|      2 | xUnit Test Patterns |     31 |
+-----+-----+-----+
2 rows in set (0.10 sec)

mysql> SELECT * FROM quotations;
+-----+-----+-----+
| Quote_ID | Quotation           | Books_Book_ID |
+-----+-----+-----+
|      1 | Programming to an Interface, not an Implementation |      1 |
|      2 | Philosophy of Test Automation |      2 |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

Let's go behind the scenes to understand the code better.

How it works...

The `GUI_SQL_class.py` code creates the database, adds tables to it, and then inserts data into the two tables we created.



Here, we outline the code without showing all the implementation details in order to preserve space because it would take too many pages to show the entire code.

We created a MySQL database, connected to it, and then created two tables that hold the data for a favorite book or journal quotation.

We distributed the data between two tables because the quotations tend to be rather large, while the book titles and book page numbers are very short. By doing this, we can increase the efficiency of our database.



In SQL database language, separating data into separate tables is called *normalization*. One of the most important things you need to do while using a SQL database is to segregate data into related tables, also known as **relationships**.

Let's move on to the next recipe.

Using the SQL UPDATE command

This recipe will use the code from the previous recipe, *Using the SQL INSERT command*, explain it in more detail, and then extend the code to update the data.

In order to update the data that we previously inserted into our MySQL database tables, we need to use the SQL UPDATE command.

Getting ready

This recipe builds on the previous recipe, *Using the SQL INSERT command*, so read and study the previous recipe in order to follow the code in this recipe, where we will modify the existing data.

How to do it...

Let's take a look at how we can use the SQL UPDATE command:

1. First, we will display the data to be modified by running the following Python to the MySQL command. Sequentially, we perform the following steps:
 1. Open `GUI_MySQL_class.py`.
 2. Look at the `showData` method:

```
import mysql.connector
import Ch07_Code.GuiDBConfig as guiConf

class MySQL():
    # class variable
    GUIDB = 'GuidB'
    #-----
    def showData(self):
        # connect to MySQL
        conn, cursor = self.connect()

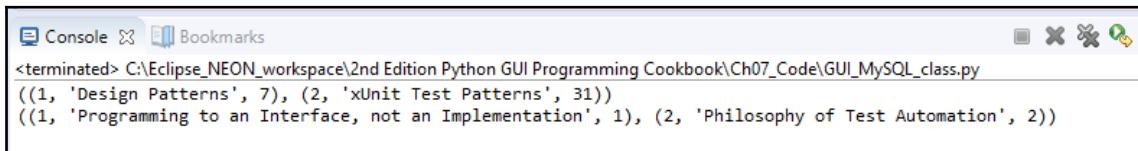
        self.useGuiDB(cursor)

        # execute command
        cursor.execute("SELECT * FROM books")
        print(cursor.fetchall())

        cursor.execute("SELECT * FROM quotations")
        print(cursor.fetchall())

        # close cursor and connection
        self.close(cursor, conn)
    #=====
    if __name__ == '__main__':
        # Create class instance
        mySQL = MySQL()
        mySQL.showData()
```

2. Running the preceding code gives us the following output:

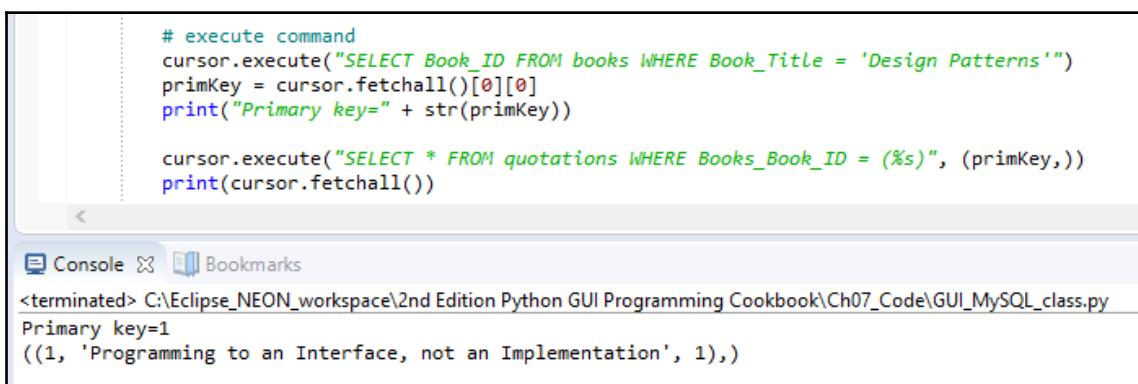


The screenshot shows the Eclipse IDE's Console view. The title bar says "Console Bookmarks". The output window displays the command "<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch07_Code\GUI_SQL_class.py" followed by the result of the code execution: "((1, 'Design Patterns', 7), (2, 'xUnit Test Patterns', 31)) ((1, 'Programming to an Interface, not an Implementation', 1), (2, 'Philosophy of Test Automation', 2))".

3. Look at the updateGOF method:

```
#-----
def updateGOF(self):
    # connect to MySQL
    conn, cursor = self.connect()
    self.useGuiDB(cursor)
    # execute command
    cursor.execute("SELECT Book_ID FROM books WHERE Book_Title =
'Design Patterns'")
    primKey = cursor.fetchall()[0][0]
    print("Primary key=" + str(primKey))
    cursor.execute("SELECT * FROM quotations WHERE Books_Book_ID =
(%s)", (primKey,))
    print(cursor.fetchall())
    # close cursor and connection
    self.close(cursor, conn)
#=====
if __name__ == '__main__':
    mySQL = MySQL()           # Create class instance
    mySQL.updateGOF()
```

4. Run the method located in GUI_SQL_class.py:



The screenshot shows the Eclipse IDE's Console view. The title bar says "Console Bookmarks". The output window displays the command "<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch07_Code\GUI_SQL_class.py" followed by the result of the code execution: "Primary key=1 ((1, 'Programming to an Interface, not an Implementation', 1),)". The code block above shows the specific SQL queries used to retrieve the primary key and the associated quotation.

5. Add the following code and run it:

```
#-----
def showDataWithReturn(self):
    # connect to MySQL
    conn, cursor = self.connect()

    self.useGuiDB(cursor)

    # execute command
    cursor.execute("SELECT Book_ID FROM books WHERE Book_Title =
'Design Patterns'")
    primKey = cursor.fetchall()[0][0]
    print(primKey)

    cursor.execute("SELECT * FROM quotations WHERE Books_Book_ID =
(%s)", (primKey,))
    print(cursor.fetchall())

    cursor.execute("UPDATE quotations SET Quotation =
(%s) WHERE Books_Book_ID = (%s)",
("Pythonic Duck Typing: If it walks like a duck and
talks like a duck it probably is a duck...", primKey))

    # commit transaction
    conn.commit ()

    cursor.execute("SELECT * FROM quotations WHERE Books_Book_ID =
(%s)", (primKey,))
    print(cursor.fetchall())

    # close cursor and connection
    self.close(cursor, conn)

#=====
if __name__ == '__main__':
    # Create class instance
    mySQL = MySQL()
    #-----
    mySQL.updateGOF()
    book, quote = mySQL.showDataWithReturn()
    print(book, quote)
```

6. Open a MySQL client window and run the `SELECT * FROM` statements:

```
mysql> USE guidb
Database changed
mysql> SELECT * FROM books;
+-----+-----+
| Book_ID | Book_Title      | Book_Page |
+-----+-----+
|      1 | Design Patterns   |        7 |
|      2 | xUnit Test Patterns |       31 |
+-----+-----+
2 rows in set (0.10 sec)

mysql> SELECT * FROM quotations;
+-----+-----+-----+
| Quote_ID | Quotation           | Books_Book_ID |
+-----+-----+-----+
|      1 | Programming to an Interface, not an Implementation |      1 |
|      2 | Philosophy of Test Automation                         |      2 |
+-----+-----+-----+

mysql> SELECT * FROM books;
+-----+-----+
| Book_ID | Book_Title      | Book_Page |
+-----+-----+
|      1 | Design Patterns   |        7 |
|      2 | xUnit Test Patterns |       31 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM quotations;
+-----+
| Quote_ID | Quotation
0 |
+-----+
|      1 | Pythonic Duck Typing: If it walks like a duck and talks like a duck it probably is a duck... |
1 |
|      2 | Philosophy of Test Automation
2 |
+-----+
|
```

Let's go behind the scenes to understand the code better.

How it works...

First, we opened `GUI_MySQL_class.py` or typed the code that was shown in our own module and ran it.

We may not agree with the *Gang of Four*, so let's change their famous programming quote.



The Gang of Four are the four authors who created the world-famous book called *Design Patterns*, which strongly influenced our entire software industry to recognize, think, and code using software *design patterns*.

We did this by updating our database of favorite quotes. First, we retrieved the primary key value by searching for the book title. Then, we passed that value into our search for the quote.

Now that we know the primary key of the quote, we can update the quote by executing the SQL UPDATE command.

Before we ran the code, our title with `Book_ID = 1` was related via a *primary key* to *foreign key* relationship to the quotation in the `Books_Book_ID` column of the quotation table. This is the original quotation from the *Design Patterns* book.

In step 5, we updated the quotation related to this ID via the SQL UPDATE command.

None of the IDs have changed, but the quotation that is now associated with `Book_ID = 1` has changed, as can be seen in the second MySQL client window.

In this recipe, we retrieved the existing data from our database and database tables that we created in the previous recipes. We inserted data into the tables and updated our data using the SQL UPDATE command.

Let's move on to the next recipe.

Using the SQL DELETE command

In this recipe, we will use the SQL DELETE command to delete the data we created in the previous recipe, *Using the SQL UPDATE command*.

While deleting data might sound trivial at first, once we get a rather large database design in production, things might not be that easy any more.

Because we have designed our GUI database by *relating* two tables via a *primary to foreign key relation*, when we delete certain data, we do not end up with *orphan records* because this database design takes care of *cascading* deletes.

Getting ready

This recipe uses the MySQL database, tables, and the data that was inserted into those tables from the previous recipe, *Using the SQL UPDATE command*. In order to demonstrate how to create orphan records, we will have to change the design of one of our database tables.



Changing the design to intentionally create a poor design is for demonstration purposes only and is not the recommended way of designing a database.

How to do it...

If we create our quotations table without a *foreign key relationship* to the books table, we can end up with orphan records. Take a look at the following steps:

1. Open `GUI_MySQL_class.py` and look at `def createTablesNoFK(self):`
...:

```
# create second Table inside DB --
# No FOREIGN KEY relation to Books Table
cursor.execute("CREATE TABLE Quotations (
    Quote_ID INT AUTO_INCREMENT,
    Quotation VARCHAR(250),
    Books_Book_ID INT,
    PRIMARY KEY (Quote_ID)
) ENGINE=InnoDB")
```
2. Run the SQL command:

```
cursor.execute("DELETE FROM books WHERE Book_ID = 1")
```

3. Run the two SELECT * commands:

```
mysql> SELECT * FROM books;
+-----+-----+
| Book_ID | Book_Title      | Book_Page |
+-----+-----+
| 2       | xUnit Test Patterns |      31    |
+-----+-----+
1 row in set (0.00 sec)

mysql> SELECT * FROM quotations;
+-----+-----+-----+
| Quote_ID | Quotation           | Books_Book_ID |
+-----+-----+-----+
| 1         | Programming to an Interface, not an Implementation | 1 |
| 2         | Philosophy of Test Automation | 2 |
+-----+-----+
2 rows in set (0.00 sec)

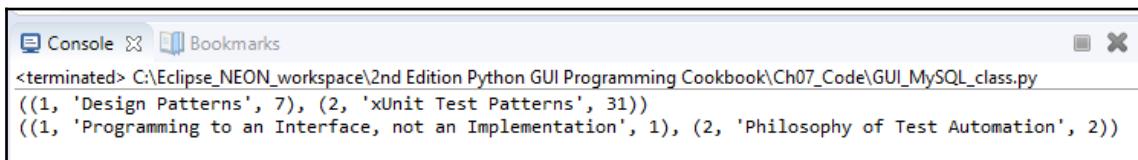
mysql>
```

4. Open `GUI_MySQL_class.py` and look at `def createTables(self): ...:`

```
# create second Table inside DB
cursor.execute("CREATE TABLE Quotations (
    Quote_ID INT AUTO_INCREMENT,
    Quotation VARCHAR(250),
    Books_Book_ID INT,
    PRIMARY KEY (Quote_ID),
    FOREIGN KEY (Books_Book_ID)
    REFERENCES Books(Book_ID)
    ON DELETE CASCADE
) ENGINE=InnoDB")

#=====
if __name__ == '__main__':
    # Create class instance
    mySQL = MySQL()
    mySQL.showData()
```

5. Run the `showData()` method:



The screenshot shows a Java IDE's console window titled "Console". It displays the command line and the output of a Python script named "GUI_MySQL_class.py". The output shows the results of a database query, specifically a list of tuples representing book data from a MySQL database.

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch07_Code\GUI_MySQL_class.py
((1, 'Design Patterns', 7), (2, 'xUnit Test Patterns', 31))
((1, 'Programming to an Interface, not an Implementation', 1), (2, 'Philosophy of Test Automation', 2))
```

6. Run the `deleteRecord()` method, followed by the `showData()` method:

```
import mysql.connector
import Ch07_Code.GuiDBConfig as guiConf

class MySQL():
    #-----
    def deleteRecord(self):
        # connect to MySQL
        conn, cursor = self.connect()

        self.useGuiDB(cursor)

        # execute command
        cursor.execute("SELECT Book_ID FROM books WHERE Book_Title
= 'Design Patterns'")
        primKey = cursor.fetchall()[0][0]
        # print(primKey)

        cursor.execute("DELETE FROM books WHERE Book_ID = (%s)",
(primKey,))

        # commit transaction
        conn.commit()

        # close cursor and connection
        self.close(cursor, conn)
    #=====
    if __name__ == '__main__':
        # Create class instance
        mySQL = MySQL()
        #-----
        mySQL.deleteRecord()
        mySQL.showData()
```

7. The preceding code results in the following output:



The screenshot shows a Eclipse IDE's Console view. The code in the console is:

```
#-  
mySQL.deleteRecord()  
mySQL.showData()
```

The output from the console is:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch07_Code\GUI_MySQL_class.py  
((2, 'xUnit Test Patterns', 31),)  
((2, 'Philosophy of Test Automation', 2),)
```

Let's go behind the scenes to understand the code better.

How it works...

We kept our database design simple by using only two database tables.

While this works when we delete data, there is always a chance of us ending up with orphan records. What this means is that we delete data in one table but somehow do not delete the related data in another SQL table. Here, we started by intentionally showing how **orphan records** can be created.

After inserting data into the `books` and `quotations` tables, if we execute a `DELETE` statement, we are only deleting the book with `Book_ID = 1`, while the related quotation with `Books_Book_ID = 1` is left behind.

This is an **orphaned record**. A book record that has a `Book_ID` of 1 no longer exists.



This situation can cause **data corruption**, which we can avoid by using *cascading* deletes.

We prevented this in the creation of the tables by adding certain database *constraints*. When we created the table that holds the quotations in a previous recipe, we created our `quotations` table with a *foreign key constraint* that explicitly references the *primary key* of the `books` table, linking the two.



The FOREIGN KEY relationship includes the ON DELETE CASCADE attribute, which basically tells our MySQL server to delete the related records in this table when the records that this foreign key relates to are deleted.

Because of this design, no *orphan* records will be left behind, which is what we want.



In MySQL, we have to specify ENGINE=InnoDB on both of the related tables in order to use *primary key* to *foreign key* relationships.

The `showData()` method shows us that we have two records that are related via *primary key* to *foreign key* relationships.

When we now delete a record in the `books` table, we expect the related record in the `quotations` table to also be deleted by a cascading delete.

After executing the commands to delete and show records, we got the new results.



The famous design patterns are gone from our database of favorite quotations. This is meant as a joke—I personally highly value the famous design patterns. However, Python's *duck typing* is a very cool feature indeed!

We triggered *cascading* deletes in this recipe by designing our database in a solid fashion via *primary key* to *foreign key* relationships with *cascading* deletes.

This keeps our data sane and integral.

In the next recipe, we will use the code of our `GUI_MySQL_class.py` module from our Python GUI.

Storing and retrieving data from our MySQL database

We will use our Python GUI to insert data into our MySQL database tables. We already refactored the GUI we built in the previous recipes in preparation for connecting and using a database.

We will use two textbox Entry widgets, into which we can type the book or journal title and the page number. We will also use a ScrolledText widget to type our favorite book quotations into, which we will then store in our MySQL database.

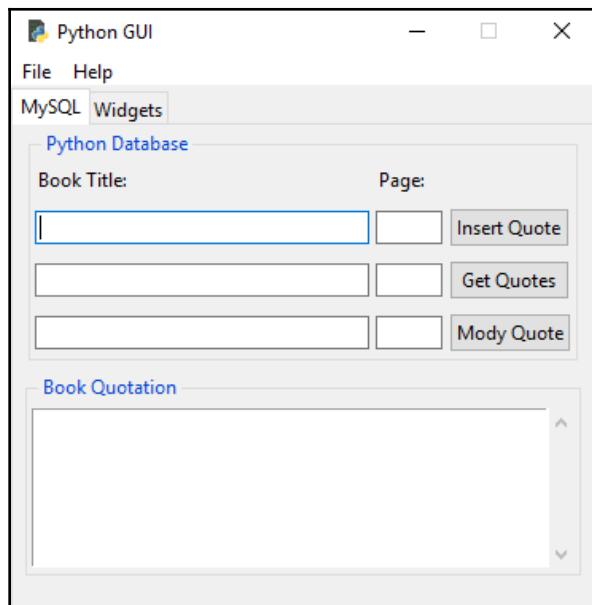
Getting ready

This recipe will build on the MySQL database and tables we created in the previous recipes of this chapter.

How to do it...

We will insert, retrieve, and modify our favorite quotations using our Python GUI. We refactored the MySQL tab of our GUI in preparation for this. Let's look at how we can deal with this:

1. Open `GUI_SQL.py`.
2. Running the code in this file shows us our GUI:

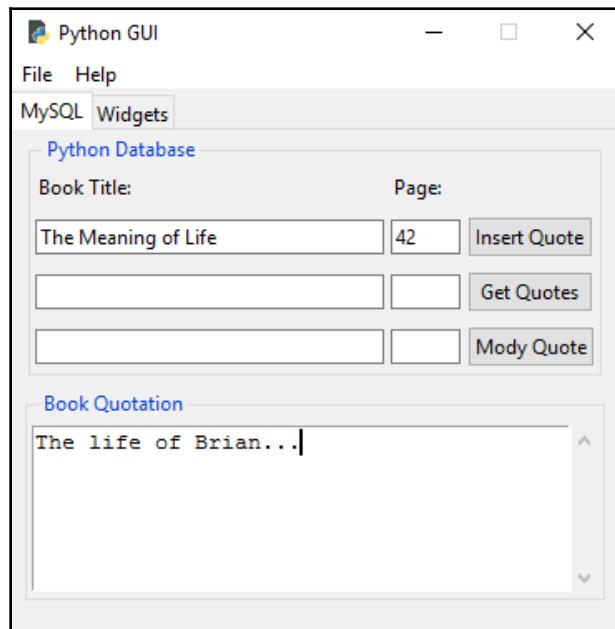


3. Open `GUI_MySQL.py`.
4. Notice the `insertQuote()` method, as shown here:

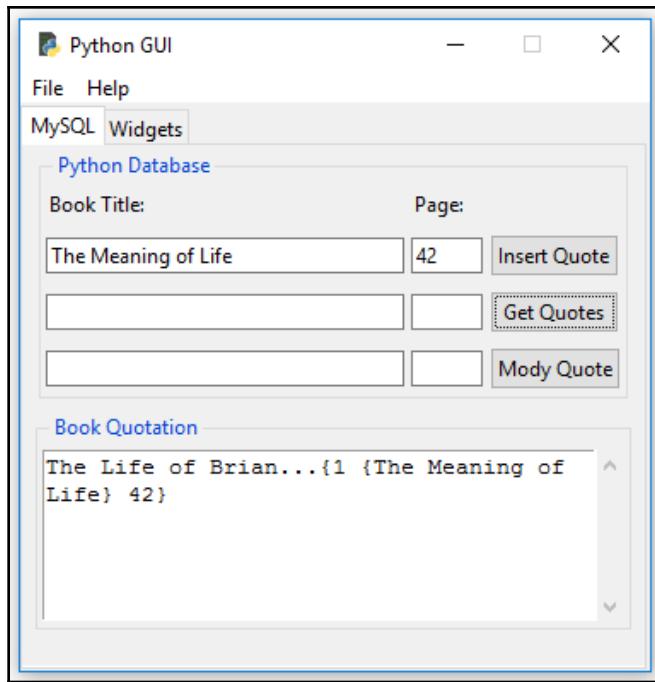
```
# Adding a Button
self.action = ttk.Button(self.mySQL, text="Insert Quote",
command=self.insertQuote)
self.action.grid(column=2, row=1)

# Button callback
def insertQuote(self):
    title = self.bookTitle.get()
    page = self.pageNumber.get()
    quote = self.quote.get(1.0, tk.END)
    print(title)
    print(quote)
    self.mySQL.insertBooks(title, page, quote)
```

5. Run `GUI_MySQL.py`, enter a quotation, and click the **Insert Quote** button:



6. Click **Get Quotes**:



7. Open `GUI_MySQL.py` and look at the `getQuote` method and button:

```
# Adding a Button
self.action1 = ttk.Button(self.mySQL, text="Get Quotes",
command=self.getQuote)
self.action1.grid(column=2, row=2)

# Button callback
def getQuote(self):
    allBooks = self.mySQL.showBooks()
    print(allBooks)
    self.quote.insert(tk.INSERT, allBooks)
```

8. Open `GUI_MySQL.py` and look at `self.mySQL` and `showBooks()`:

```
from Ch07_Code.GUI_MySQL_class import MySQL
class OOP():
    def __init__(self):
        # create MySQL instance
        self.mySQL = MySQL()
```

```
class MySQL():
    -----
    def showBooks(self):
        # connect to MySQL
        conn, cursor = self.connect()

        self.useGuiDB(cursor)

        # print results
        cursor.execute("SELECT * FROM Books")
        allBooks = cursor.fetchall()
        print(allBooks)

        # close cursor and connection
        self.close(cursor, conn)

    return allBooks
```

Let's go over how this recipe works.

How it works...

In order to make the buttons in `GUI_SQL.py` do something, we connect them to *callback functions*, like we have done many times in this book. We display the data in the `ScrolledText` widget, below the buttons.

In order to do this, we import the `GUI_SQL_class.py` module. The entire code that talks to our MySQL server instance and database resides in this module, which is a form of *encapsulating* the code in the spirit of **object-oriented programming (OOP)**.

We connect the **Insert Quote** button to the `insertQuote()` method callback.

When we run our code, we can insert data from our Python GUI into our MySQL database.

After entering a book title and book page, as well as a quote from the book, we *insert* the data into our database by clicking the **Insert Quote** button.



Our current design allows for titles, pages, and a quotation. We can also insert our favorite quotations from movies. While a movie does not have pages, we can use the page column to insert the approximate time when the quotation occurred within the movie.

After inserting the data, we verified that it made it into our two MySQL tables by clicking the **Get Quotes** button, which then displayed the data we inserted into our two MySQL database tables, as shown in the screenshot in *step 6*.

Clicking the **Get Quotes** button invokes the callback method we associated with the button click event. This gives us the data that we display in our `ScrolledText` widget.

We used the `self.mysql` class instance attribute to invoke the `showBooks()` method, which is a part of the `MySQL` class we imported.

In this recipe, we imported the Python module we wrote, which contains all of the coding logic that we need so that we can connect to our MySQL database. It also knows how to insert, update, and delete data.

Now, we have connected our Python GUI to this SQL logic.

Let's move on to the next recipe.

Using MySQL Workbench

MySQL has a very nice GUI that we can download for free. It's called **MySQL Workbench**.

In this recipe, we will successfully install Workbench and then use it to run SQL queries against the GuiDB we created in the previous recipes.

Getting ready

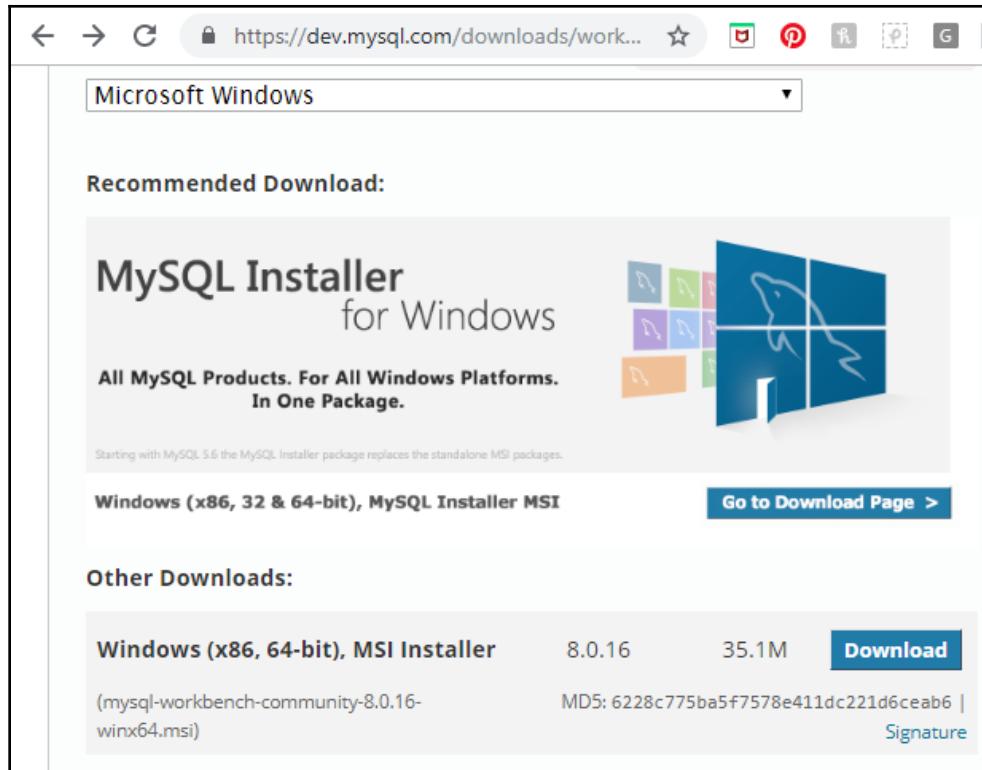
In order to use this recipe, you will need MySQL database we developed in the previous recipes. You will also need a running MySQL server.

How to do it...

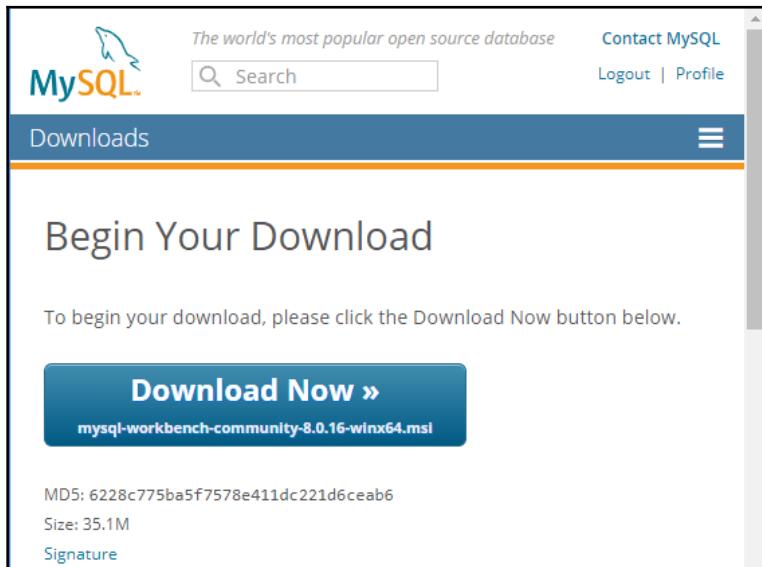
We can download MySQL Workbench from the official MySQL website: <https://dev.mysql.com/downloads/workbench/>.

Let's look at how we can perform this recipe:

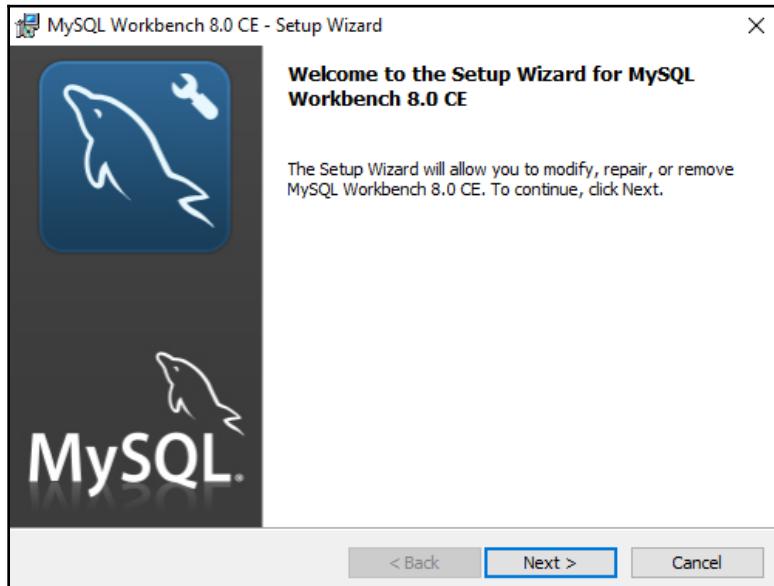
1. Download the MySQL Workbench installer.
2. Click the **Download** button:



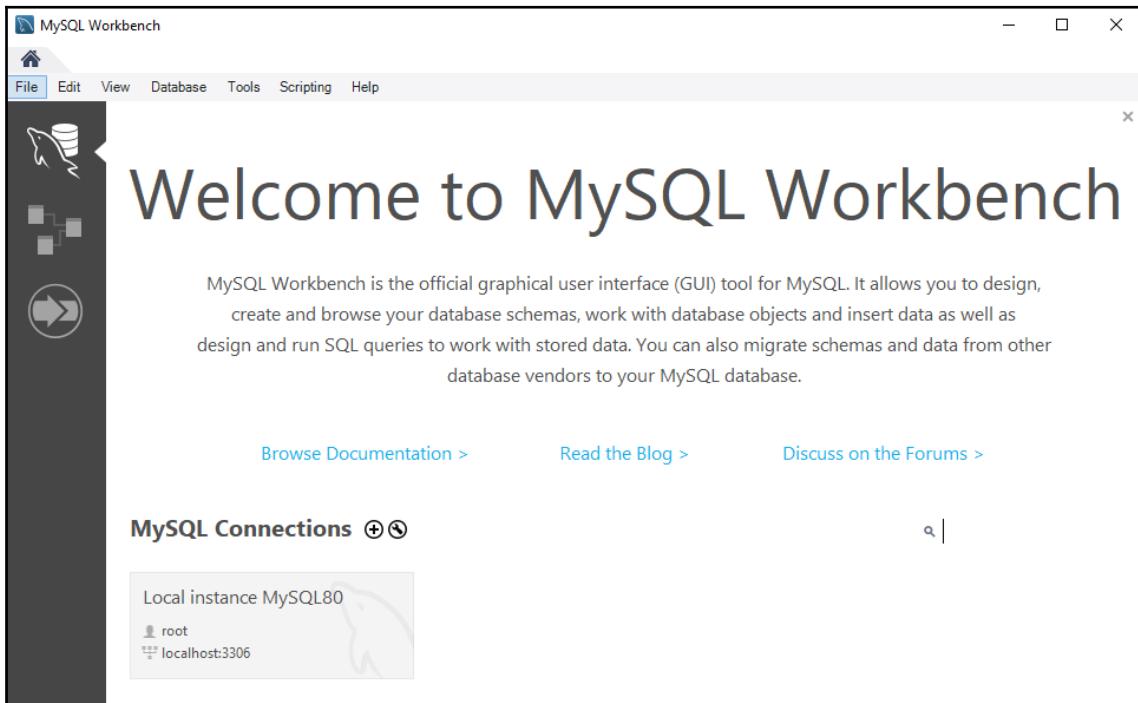
3. Run the installation:



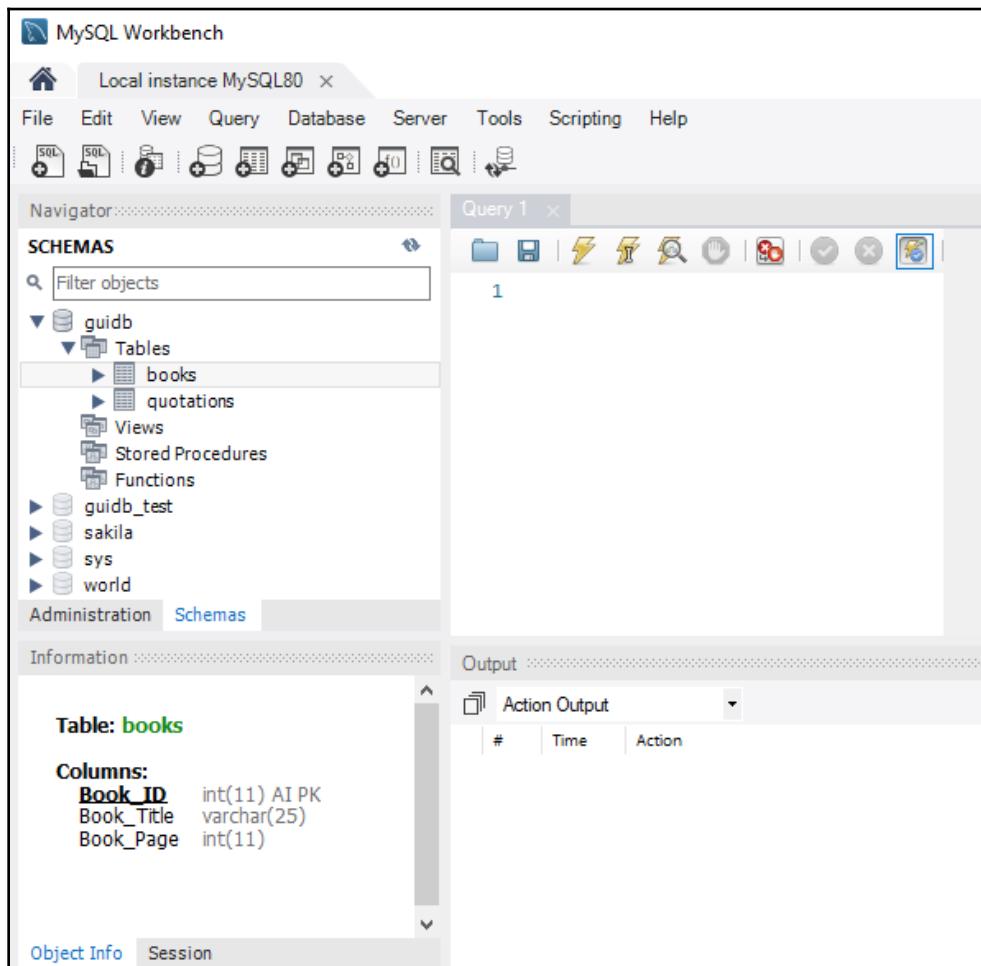
4. Click **Next >** until the installation is complete:



5. Open MySQL Workbench:



6. Select our guidb:



7. Write and execute some SQL commands:

The screenshot shows the MySQL Workbench interface. At the top, there's a toolbar with various icons for database management. Below it is a menu bar with File, Edit, View, Query, Database, Server, Tools, Scripting, and Help. The main area is titled "Query 1" and contains two SQL statements:

```
2 •  SELECT * FROM books;
3 •  SELECT * FROM quotations;
```

Below the queries is a "Result Grid" table with the following data:

Quote_ID	Quotation	Books_Book_ID
1	Programming to an Interface, not an Implement...	1
2	Programming to an Interface, not an Implement...	2
3	Philosophy of Test Automation	3
*	NULL	NULL

At the bottom, there are tabs for "books 1" and "quotations 2". On the right side, there's a sidebar with a "Result Grid" button. The "Action Output" section shows the execution log:

#	Time	Action	Message	Duration / Fetch
1	06:51:44	use guidb	0 row(s) affected	0.000 sec
2	06:51:44	SELECT * FROM books LIMIT 0, 1000	3 row(s) returned	0.000 sec / 0.000 sec
3	06:51:44	SELECT * FROM quotations LIMIT 0, 1000	3 row(s) returned	0.000 sec / 0.000 sec

Let's go behind the scenes to understand the code better.

How it works...

When you installed MySQL, if you had the required components already installed on your PC, you may already have MySQL Workbench installed. If you do not have Workbench installed, *steps 1 to 3* show you how to install MySQL Workbench.



MySQL Workbench is a GUI in itself, very similar to the one we developed in the previous recipes. It does come with some additional features that are specific to working with MySQL. 8.0 CE in the installer window is an abbreviation for version **8.0 Community Edition**.

When you start up MySQL Workbench, it will prompt you to connect. Use the **root** user and password you created for it. MySQL Workbench is smart enough to recognize that you're running a MySQL server and the port it is listening on.

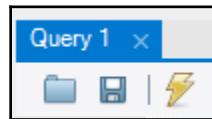
Once you are successfully logged in to your MySQL server instance, you can select the `guidb` we created.

We can find our `guidb` underneath the **SCHEMAS** label.



In some literature and products, databases are often called **SCHEMAS**. Schematics refer to the structure and layout of a database. Personally, coming from Microsoft SQL Server, I am used to referring to them simply as *databases*.

We can type SQL commands into the **Query Editor** and execute our commands by clicking the lightning bolt icon. It is the button toward the top right, as shown in the following screenshot:



We can see the results in the **Result Grid**. We can click on the different tabs to see the different results.

Now, we can connect to our MySQL database via the MySQL Workbench GUI. We can execute the same SQL commands we issued before and get the same results that we did when we executed them in our Python GUI.

There's more...

With the knowledge we have gained throughout the recipes within this and the preceding chapters, we are now well positioned to create our own GUIs written in Python, all of which can connect and talk to MySQL databases.

8

Internationalization and Testing

In this chapter, we will *internationalize* our GUI by displaying text on labels, buttons, tabs, and other widgets, in different languages. We will start simply and then explore how we can prepare our GUI for internationalization at the design level.

We will also *localize* the GUI, which is slightly different from internationalization.



As these words are long, they have been abbreviated to use the first character of the word, followed by the total number of characters in between the first and last character, followed by the last character of the word. So, **internationalization** becomes **I18N**, and **localization** becomes **L10N**.

We will also test our GUI code, write unit tests, and explore the value unit tests can provide in our development efforts, which will lead us to the best practice of refactoring our code.

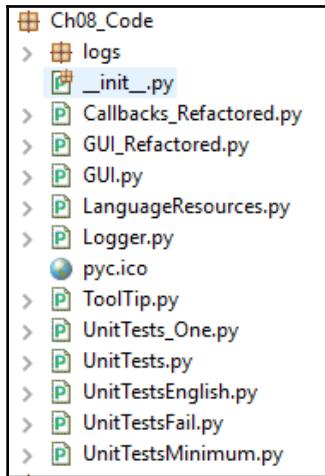


There are no additional Python packages to install. We write our own code in Python and the unit testing framework is shipped with Python, so we can simply import it.

Knowing how to both internationalize and test our code is an essential skill every programmer needs to know.

In this chapter, you will acquire the valuable skills of testing, refactoring, and internationalization.

Here is an overview of the Python modules for this chapter:



We will internationalize and test our Python GUI, covering the following recipes:

- Displaying widget text in different languages
- Changing the entire GUI language all at once
- Localizing the GUI
- Preparing the GUI for internationalization
- How to design a GUI in an agile fashion
- Do we need to test the GUI code?
- Setting debug watches
- Configuring different debug output levels
- Creating self-testing code using Python's `__main__` section
- Creating robust GUIs using unit tests
- How to write unit tests using the Eclipse PyDev **Integrated Development Environment (IDE)**

Displaying widget text in different languages

The easiest way to internationalize text strings in Python is by moving them into a separate Python module and then selecting the language to be displayed in our GUI by passing in an argument to this module.

While this approach, according to online search results, is not highly recommended, depending on the specific requirements of the application you are developing, it may still be the most pragmatic and fastest to implement.

Getting ready

We will reuse the Python GUI we created earlier in *Chapter 7, Storing Data in Our MySQL Database via Our GUI*. We will comment out one line of Python code that creates the MySQL tab because we do not interact with a MySQL database in this chapter.

How to do it...

In this recipe, we will start the I18N of our GUI by changing the window's title from English to another language.

As the name *GUI* is the same in other languages, we will first expand the name that enables us to see the visual effects of our changes. Let's now see the steps in detail:

1. Open `GUI_MySQL.py` from the previous chapter (*Chapter 7, Storing Data in Our MySQL Database via Our GUI*) and save it as `GUI.py`.

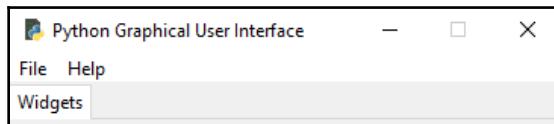
The following was our previous line of code:

```
self.win.title("Python GUI")
```

2. Let's change this to the following code. Also, comment out the creation of the MySQL tab:

```
self.win.title("Python Graphical User Interface")      # new window
title
# self.mysql = MySQL()                                # comment
this line out
```

3. The preceding code change results in the following title for our GUI program:



Please note that, in this chapter, we will use English and German to exemplify the principle of internationalizing our Python GUI.

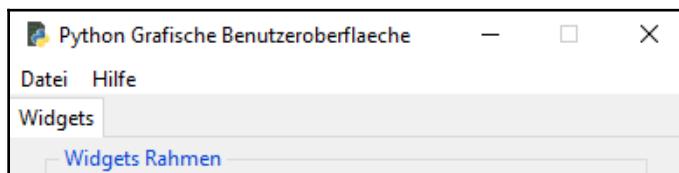
4. Let's create a new Python module and name it `LanguageResources.py`. Let's next move the English string of our GUI title into this module and also create a German version.
5. Add the following code:

```
class I18N():  
    '''Internationalization'''  
    def __init__(self, language):  
        if language == 'en': self.resourceLanguageEnglish()  
        elif language == 'de': self.resourceLanguageGerman()  
        else: raise NotImplementedError('Unsupported language.')  
  
    def resourceLanguageEnglish(self):  
        self.title = "Python Graphical User Interface"  
  
    def resourceLanguageGerman(self):  
        self.title = 'Python Grafische Benutzeroberflaeche'
```

6. Import the `I18N` class and change the language to '`de`':

```
from Ch08_Code.LanguageResources import I18N  
class OOP():  
    def __init__(self, language='en'):  
        self.win = tk.Tk() # Create instance  
        self.i18n = I18N('de') # Select different  
        language  
        self.win.title(self.i18n.title) # Add a title using  
        self.i18n
```

7. Running the preceding code, we now get the following internationalized result:



Let's now go behind the scenes to understand the code better.

How it works...

Starting from *step 4*, we break out the hardcoded strings that are part of our GUI into their own separate module, `LanguageResources.py`. Within the class's `__init__()` method, we select which language our GUI will display, depending on the passed-in language argument. We then import the `LanguageResources.py` module into our OOP class module.

We set the default language to '`en`', which means English.

Within `GUI.py`, we are creating an instance of the `I18N` class. This class resides in `LanguageResources.py`, so the name of our class is shorter and different to the name of the Python module. We save the selected language in a class instance attribute, `self.i18n`, and use it to display the title. We are separating the GUI from the languages it displays, which is an OOP design principle.

We can further modularize our code by separating the internationalized strings into separate files, potentially in XML or another format. We could also read them from a MySQL database.

This is a **separation of concerns (SoC)** coding approach, which is at the heart of OOP programming.

Changing the entire GUI language all at once

In this recipe, we will change all of the GUI display names, all at once, by refactoring all the previously hardcoded English strings into a separate Python module and then internationalizing those strings.

This recipe shows that it is a good design principle to avoid hardcoding any strings, that our GUI displays, but to separate the GUI code from the text that is displayed by the GUI.

Designing our GUI in a modular way makes internationalizing it much easier.

Getting ready

We will continue to use the GUI from the previous recipe, `GUI.py`. In that recipe, we had already internationalized the title of the GUI. We will enhance the `LanguageResources.py` module from the previous recipe as well by adding more internationalized strings.

How to do it...

In order to internationalize the text displayed in all of our GUI widgets, we have to move all hardcoded strings into a separate Python module, and this is what we'll do next:

1. Open the `LanguageResources.py` module.
2. Add the following code for the English internationalized strings:

```
class I18N():
    '''Internationalization'''
    def __init__(self, language):
        if language == 'en': self.resourceLanguageEnglish()
        elif language == 'de': self.resourceLanguageGerman()
        else: raiseNotImplementedError('Unsupported language.')

    def resourceLanguageEnglish(self):
        self.title = "Python Graphical User Interface"

        self.file = "File"
        self.new = "New"
        self.exit = "Exit"
        self.help = "Help"
        self.about = "About"

        self.WIDGET_LABEL = ' Widgets Frame '

        self.disabled = "Disabled"
        self.unChecked = "UnChecked"
        self.toggle = "Toggle"

        # Radiobutton list
        self.colors = ["Blue", "Gold", "Red"]
        self.colorsIn = ["in Blue", "in Gold", "in Red"]

        self.labelsFrame = ' Labels within a Frame '
        self.chooseNumber = "Choose a number:"
        self.label2 = "Label 2"

        self.mgrFiles = ' Manage Files '

        self.browseTo = "Browse to File..."
        self.copyTo = "Copy File To : "
```

3. In the Python GUI .py module, replace all the hardcoded strings with an instance of our new I18N class, for example, self.i18n.colorsIn:

```
from Ch08_Code.LanguageResources import I18N
class OOP():
    def __init__(self, language='en'):
        self.win = tk.Tk() # Create instance
        self.i18n = I18N('de') # Select language
        self.win.title(self.i18n.title) # Add a title

    # Radiobutton callback function
    def radCall(self):
        radSel = self.radVar.get()
        if radSel == 0: self.widgetFrame.configure(text=
            self.i18n.WIDGET_LABEL +
            self.i18n.colorsIn[0])
        elif radSel == 1: self.widgetFrame.configure(text=
            self.i18n.WIDGET_LABEL +
            self.i18n.colorsIn[1])
        elif radSel == 2: self.widgetFrame.configure(text=
            self.i18n.WIDGET_LABEL +
            self.i18n.colorsIn[2])
```

We can now implement the translation to German by simply filling in the variable names with the corresponding words.

4. Add the following code to LanguageResources.py:

```
class I18N():
    '''Internationalization'''
    def __init__(self, language):
        if language == 'en': self.resourceLanguageEnglish()
        elif language == 'de': self.resourceLanguageGerman()
        else: raise NotImplementedError('Unsupported language.')

    def resourceLanguageGerman(self):
        self.file = "Datei"
        self.new = "Neu"
        self.exit = "Schliessen"
        self.help = "Hilfe"
        self.about = "Ueber"

        self.WIDGET_LABEL = ' Widgets Rahmen '

        self.disabled = "Deaktiviert"
        self.unChecked = "Nicht Markiert"
        self.toggle = "Markieren"
```

```

# Radiobutton list
self.colors    = ["Blau", "Gold", "Rot"]
self.colorsIn = ["in Blau", "in Gold", "in Rot"]

self.labelsFrame = ' Etiketten im Rahmen '
self.chooseNumber = "Waehle eine Nummer:"
self.label2      = "Etikette 2"

self.mgrFiles = ' Dateien Organisieren '

self.browseTo = "Waehle eine Datei... "
self.copyTo   = "Kopiere Datei zu : "

```

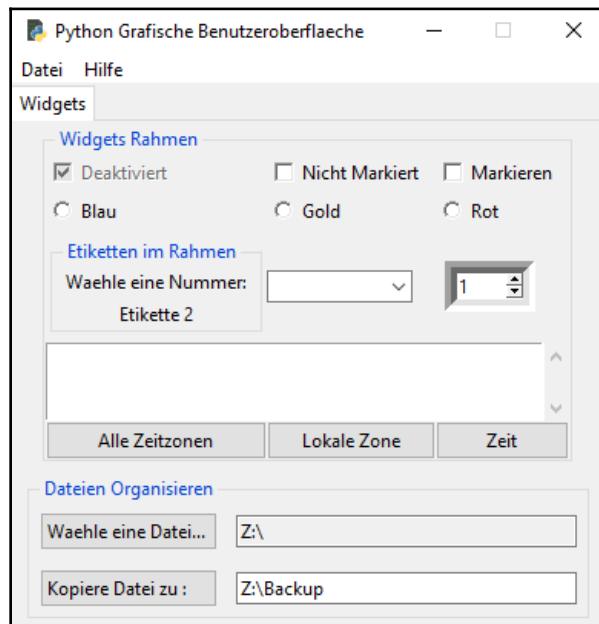
5. In our GUI code, we can now change the entire GUI display language in one line of Python code:

```

classOOP () :
    def __init__(self, language='en'):
        self.win = tk.Tk()           # Create instance
        self.i18n = I18N('de')       # Pass in language

```

6. Running the preceding code creates the following internationalized GUI:



Let's now see how this recipe works!

How it works...

In order to internationalize our GUI, we refactored hardcoded strings into a separate module and then used the same class instance attributes to internationalize our GUI by passing in a string to the initializer of our `I18N` class, effectively controlling the language our GUI displays.

Note how all of the previously hardcoded English strings have been replaced by calls to the instance of our new `I18N` class. One example is `self.win.title(self.i18n.title)`.

What this gives us is the ability to internationalize our GUI. We simply have to use the same variable names and combine them by passing in a parameter to select the language we wish to display.

We could change languages on the fly as part of the GUI as well, or we could read the local PC settings and decide which language our GUI text should display according to those settings. An example of how to read the local settings is covered in the next recipe, *Localizing the GUI*.

Previously, every single string of every widget, including the title of our GUI, the tab control names, and so on, were all hardcoded and intermixed with the code that creates the GUI.

It is a good idea to think about how we can best internationalize our GUI at the design phase of our GUI software development process.

In this recipe, we internationalized all strings displayed in our GUI widgets. We are not internationalizing the text entered into our GUI, because this depends on the local settings on your PC.

Localizing the GUI

After the first step of internationalizing our GUI, the next step is to *localize* it. Why would we wish to do this?

Well, here in the United States of America, we are all cowboys and we live in different time zones.

While we are internationalized to the US, our horses do wake up in different time zones (and do expect to be fed according to their own inner horse time zone schedule).

This is where *localization* comes in.

Getting ready

We are extending the GUI we developed in the previous recipe by localizing it.

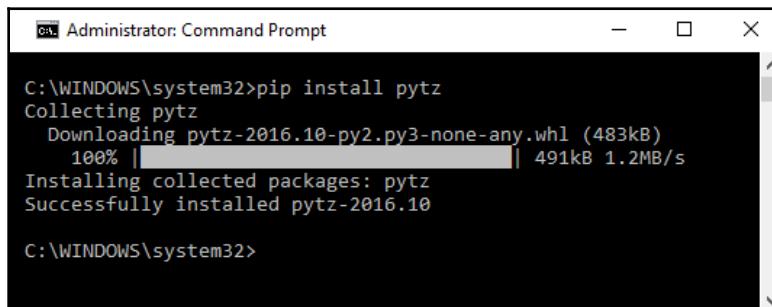
How to do it...

Let's see how to perform this recipe:

1. We start by first installing the Python `pytz` time zone module, using `pip`.
2. Next, open Command Prompt and type the following command:

```
pip install pytz
```

3. When the installation is successful, we get the following result:



```
C:\WINDOWS\system32>pip install pytz
Collecting pytz
  Downloading pytz-2016.10-py2.py3-none-any.whl (483kB)
    100% |████████████████████████████████| 491kB 1.2MB/s
Installing collected packages: pytz
Successfully installed pytz-2016.10

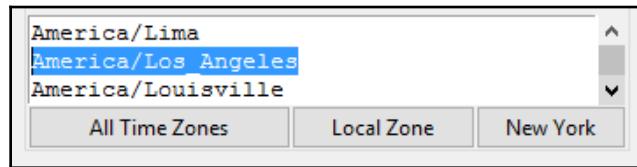
C:\WINDOWS\system32>
```

4. Next, we add a new `Button` widget to `GUI.py`. We can list all the existing time zones by running the following code, which will display the time zones in our `ScrolledText` widget by adding the `allTimeZones` method as follows:

```
import pytz
class OOP():
    # TZ Button callback
    def allTimeZones(self):
        for tz in pytz.all_timezones:
            self.scr.insert(tk.INSERT, tz + '\n')

    def createWidgets(self):
        # Adding a TZ Button
        self.allTzs = ttk.Button(self.widgetFrame,
                               text=self.i18n.timeZones,
                               command=self.allTimeZones)
        self.allTzs.grid(column=0, row=9, sticky='WE')
```

5. Clicking our new Button widget results in the following output:



6. Install the `tzlocal` Python module using `pip`, and then we can display our current locale by adding the `localZone` method and connecting it to a new Button command:

```
# TZ Local Button callback
def localZone(self):
    from tzlocal import get_localzone
    self.scr.insert(tk.INSERT, get_localzone())

def createWidgets(self):
    # Adding local TZ Button
    self.localTZ = ttk.Button(self.widgetFrame,
        text=self.i18n.localZone, command=self.localZone)
    self.localTZ.grid(column=1, row=9, sticky='WE')
```

We internationalize the strings of our two new buttons in `LanguageResources.py`.

The English version is as follows:

```
self.timeZones = "All Time Zones"
self.localZone = "Local Zone"
```

The German version is as follows:

```
self.timeZones = "Alle Zeitzonen"
self.localZone = "Lokale Zone"
```

Clicking our new button now tells us which time zone we are in (hey, we didn't know that, did we...):



7. We can now change our local time to US EST by first converting it to **Coordinated Universal Time (UTC)** and then applying the time zone function from the imported `pytz` module. Next, add the following code to `GUI.py`:

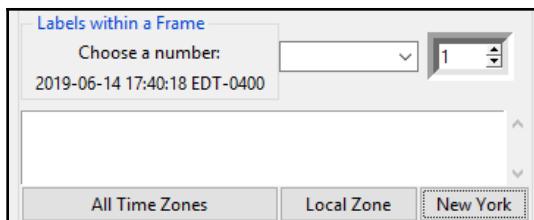
```
import pytz
class OOP():
    # Format local US time with TimeZone info
    def getDateTime(self):
        fmtStrZone = "%Y-%m-%d %H:%M:%S %Z%z"
        # Get Coordinated Universal Time
        utc = datetime.now(timezone('UTC'))
        print(utc.strftime(fmtStrZone))

        # Convert UTC datetime object to Los Angeles TimeZone
        la = utc.astimezone(timezone('America/Los_Angeles'))
        print(la.strftime(fmtStrZone))

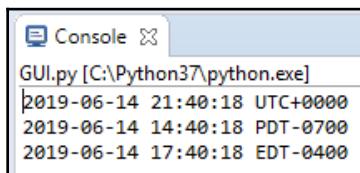
        # Convert UTC datetime object to New York TimeZone
        ny = utc.astimezone(timezone('America/New_York'))
        print(ny.strftime(fmtStrZone))

        # update GUI label with NY Time and Zone
        self.lbl2.set(ny.strftime(fmtStrZone))           # <-- set
Label 2
```

8. Clicking the button, now renamed as **New York**, results in the following output in label 2 in the top-left corner of the GUI:



9. Notice the following output in the console:



Let's learn about this recipe in the next section.

How it works...

First, we installed the Python `pytz` module using `pip`.



In this book, we are using Python 3.7, which comes with the `pip` module built in. If you are using an older version of Python, then you may have to install the `pip` module first.

The screenshot in *step 2* shows that the `pip` command downloaded the `.whl` format. If you have not done so, you might have to install the Python wheel module as well.

This installed the Python `pytz` module into the `site-packages` folder, so now we can import this module from our Python GUI code.

In order to localize date and time information, we first need to convert our local time to UTC time. We then apply the time zone information and use the `astimezone` function from the `pytz` Python time zone module to convert to any time zone across the globe!

We installed the Python `tzlocal` module using `pip`, and now we can translate our local time to a different time zone. We used US EST as an example.

In *step 8*, we converted the local time of the US west coast to UTC and then displayed the US east coast time in label 2 (`self.lbl2`) of our GUI.

At the same time, we are printing the UTC times of the cities Los Angeles and New York with their respective time zone conversions, relative to the UTC time to the console, using a US date formatting string.



UTC never observes **Daylight Saving Time (DST)**. During **Eastern Daylight Time (EDT)**, UTC is four hours ahead and, during standard time (EST), it is five hours ahead of the local time.

Preparing the GUI for internationalization

In this recipe, we will prepare our GUI for internationalization by realizing that not all is as easy as could be expected when translating English into foreign languages.

We still have one problem to solve, which is how to properly display non-English Unicode characters from foreign languages.

You might expect that displaying the German ä, ö, and ü Unicode umlaut characters would be handled by Python 3.7 automatically, but this is not the case.

Getting ready

We will continue to use the Python GUI we developed in recent chapters. First, we will change the default language to German in the `GUI.py` initialization code.

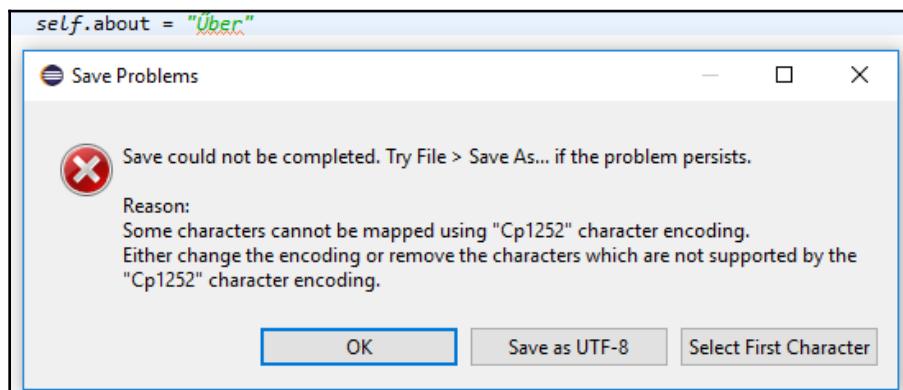
This recipe might be specific to the Eclipse PyDev IDE, but it is good to see this as an example.

How to do it...

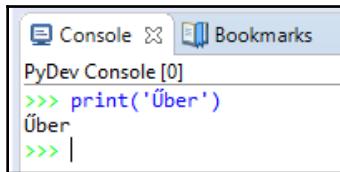
Before diving deep into the recipe, we should know that when we change the word "Ueber" to the correct German word "Über" using the umlaut character, the Eclipse PyDev plugin is not too happy.

Let's now see the steps of this recipe sequentially:

1. Open `GUI.py`.
2. Uncomment the line `self.i18n = I18N('de')` in order to use German.
3. Run `GUI.py`:

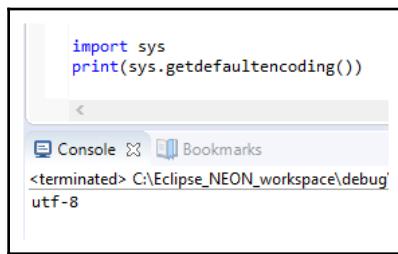


4. We get an error message, which is a little bit confusing because, when we run the same line of code from within the Eclipse PyDev Console, we get the expected result:



```
Console Bookmarks
PyDev Console [0]
>>> print('Über')
Über
>>> |
```

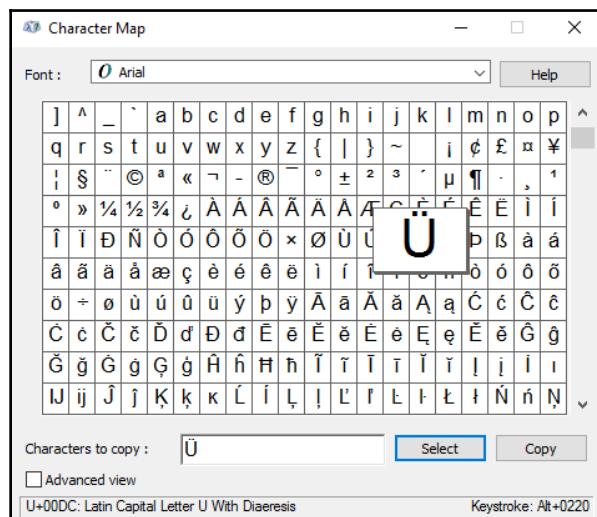
5. When we ask for the Python default encoding, we get the expected result, which is **utf-8**:



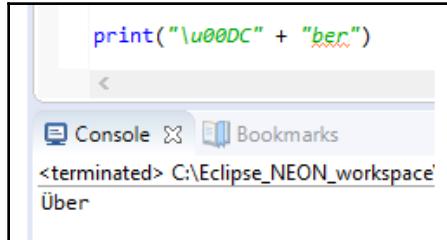
```
import sys
print(sys.getdefaultencoding())

<
Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\debug
utf-8
```

6. Using Windows built-in character map, we can find the Unicode representation of the umlaut character, which is **U+00DC** for the capital U with an umlaut:



7. While this workaround is truly ugly, it does the trick. Instead of typing in the literal character Ü, we can pass in the Unicode of U+00DC to get this character correctly displayed in our GUI:



A screenshot of an Eclipse Console window. The code in the editor is:

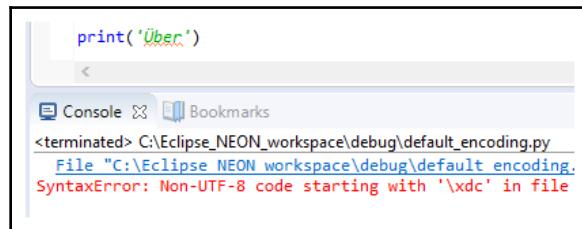
```
print("\u00DC" + "ber")
```

The console output shows:

```
<terminated> C:\Eclipse_NEON_workspace'Über'
```

8. We can also just accept the change in the default encoding from Cp1252 to UTF-8 using PyDev with Eclipse, but we may not always get the prompt to do so.

Instead, we might see the following error message displayed:



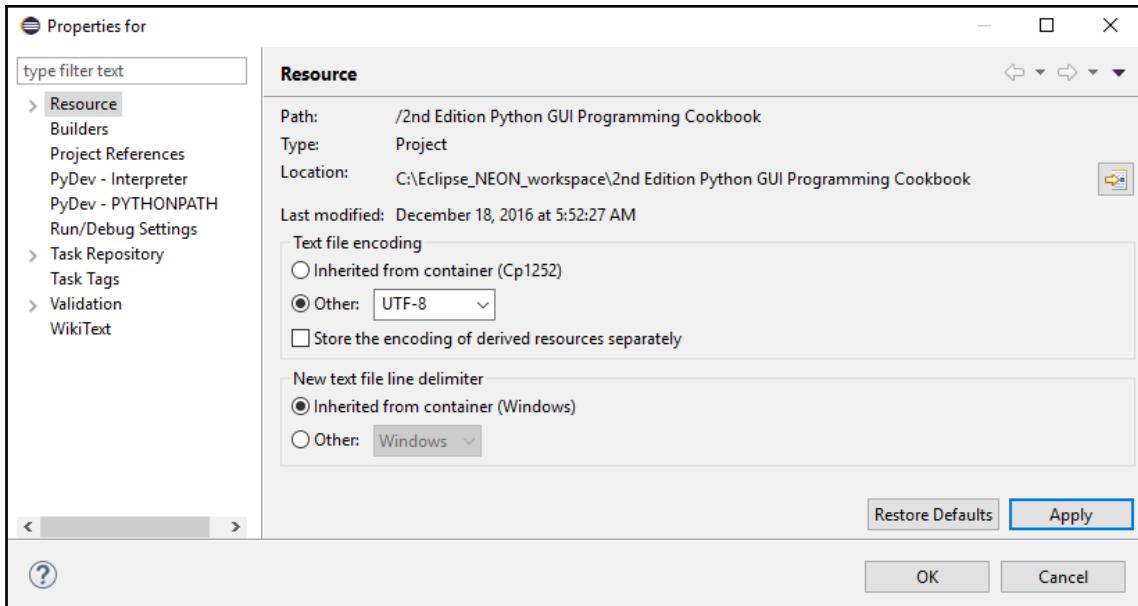
A screenshot of an Eclipse Console window. The code in the editor is:

```
print('Über')
```

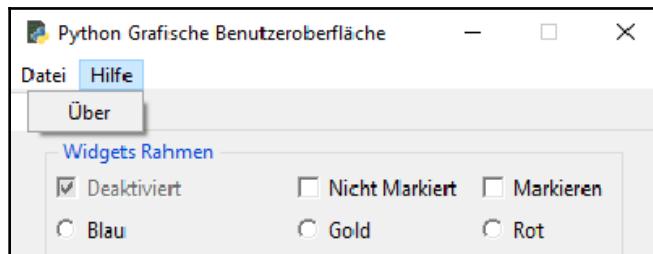
The console output shows:

```
<terminated> C:\Eclipse_NEON_workspace\debug\default_encoding.py
File "C:\Eclipse_NEON_workspace\debug\default_encoding.py"
SyntaxError: Non-UTF-8 code starting with '\xdc' in file
```

9. The way to solve this problem is to change the PyDev project's **Text file encoding** setting to **UTF-8**:



- After changing the PyDev default encoding, we now can display those German umlaut characters. We also updated the title to use the correct German ä character, GUI_.py:



Let's now go behind the scenes to understand the code better.

How it works...

Internationalization and working with foreign language Unicode characters is often not as straightforward as we would wish.

Sometimes, we have to find workarounds, and expressing Unicode characters via Python by using the direct representation by prepending \u can do the trick.

The Windows built-in character map shows us "U+00DC", which we translate into Python as "\u00DC".

We can always resort to the direct representation of Unicode.

At other times, we just have to find the settings of our development environment to adjust. We saw an example of how to do this using the Eclipse IDE.

How to design a GUI in an agile fashion

The modern agile software development approach to design and coding came out of the lessons learned by software professionals. This method applies to a GUI as much as to any other code. One of the main keys of agile software development is the continuously applied process of refactoring.

One practical example of how refactoring our code can help us in our software development work is by first implementing some simple functionality using functions.

As our code grows in complexity, we might want to refactor our functions into methods of a class. This approach would enable us to remove global variables and also to be more flexible about where we place the methods inside the class.

While the functionality of our code has not changed, the structure has.

In this process, we code, test, refactor, and then test again. We do this in short cycles and often start with the minimum code required to get some functionality to work.



Test-driven software development is one particular style of the agile development methodology.

While our GUI is working nicely, our main `GUI.py` code has been ever-increasing in complexity, and it has started to get a little bit harder to maintain an overview of our code. This means that we need to refactor our code.

Getting ready

We will refactor the GUI we created in previous chapters. We will use the English version of the GUI.

How to do it...

We have already broken out all the names our GUI displays when we internationalized it in the previous recipe. That was an excellent start to refactoring our code:

1. Let's rename our `GUI.py` file as `GUI_Refactored.py`.
2. Group the `import` statements as follows:

```
#=====
# imports
#=====
import tkinter as tk
from tkinter import ttk, scrolledtext, Menu, Spinbox, filedialog as fd, messagebox as mBox
from queue import Queue
from os import path
from Ch08_Code.ToolTip import ToolTip
from Ch08_Code.LanguageResources import I18N
from Ch08_Code.Logger import Logger, LogLevel

# Module level GLOBALS
GLOBAL_CONST = 42
```

We can refactor our code further by breaking out the callback methods into their own module.

3. Create a new module, `Callbacks_Refactored.py`.
4. In `GUI_Refactored.py`, import the `Callbacks` class.
5. In `Callbacks_Refactored.py`, add `self.callbacks = Callbacks(self)`:

```
#=====
# imports
#=====
import tkinter as tk
from tkinter import ttk, scrolledtext, Menu, Spinbox,
                    filedialog as fd, messagebox as mBox
from queue import Queue
from os import path
import Ch08_Code.ToolTip as tt
```

```
from Ch08_Code.LanguageResources import I18N
from Ch08_Code.Logger import Logger, LogLevel
from Ch08_Code.Callbacks_Refactored import Callbacks      # <-- import the class

# Module level GLOBALS
GLOBAL_CONST = 42

class OOP():
    def __init__(self):
        # Callback methods now in different module
        self.callBacks = Callbacks(self)                      # <-- pass
    in self
```

6. Add the following code to the Callbacks class:

```
#=====
# imports
#=====
import tkinter as tk
from time import sleep
from threading import Thread
from pytz import all_timezones, timezone
from datetime import datetime

class Callbacks():
    def __init__(self, oop):
        self.oop = oop

    def defaultFileEntries(self):
        self.oop.fileEntry.delete(0, tk.END)
        self.oop.fileEntry.insert(0, 'Z:')           # bogus path
        self.oop.fileEntry.config(state='readonly')
        self.oop.netwEntry.delete(0, tk.END)
        self.oop.netwEntry.insert(0, 'Z:Backup')     # bogus path

    # Combobox callback
    def _combo(self, val=0):
        value = self.oop.combo.get()
        self.oop.scr.insert(tk.INSERT, value + '\n')

    ...
```

Let's now see how this recipe works!

How it works...

We have first improved the readability of our code by grouping the related `import` statements.

By simply grouping related imports, we can reduce the number of lines of code, which improves the readability of our imports, making them appear less overwhelming.

We next broke out the callback methods into their own class and module, `Callbacks_Refactored.py`, in order to reduce the complexity of our code further.

In the initializer of our new class, the passed-in GUI instance is saved under the name `self.oop` and used throughout this new Python class module.

Running the refactored GUI code still works as before. We have only increased its readability and reduced the complexity of our code in preparation for further development work.

We had already taken the same OOP approach by having the `ToolTip` class reside in its own module, and by internationalizing all GUI strings in the previous recipes. In this recipe, we went one step further in refactoring by passing our own instance into the callback method's class that our GUI relies upon. This enables us to use all of our GUI widgets.

Now that we better understand the value of a modular approach to software development, we will most likely start with this approach in our future software designs.



Refactoring is the process of improving the structure, readability, and maintainability of the existing code. We are not adding new functionality.

Do we need to test the GUI code?

Testing our software is an important activity during the coding phase as well as when releasing service packs or bug fixes.

There are different levels of testing. The first level is developer testing, which often starts with the compiler or interpreter not letting us run buggy code, forcing us to test small parts of our code on the level of individual methods.

This is the first level of defense.

A second level of coding defensively is when our source code control system tells us about some conflicts to be resolved and does not let us check our modified code.

This is very useful and absolutely necessary when we work professionally in a team of developers. The source code control system is our friend and points out changes that have been committed to a particular branch or top-of-tree, either by ourselves or by our other developers, and tells us that our local version of the code is both outdated and has some conflicts that need to be resolved before we can submit our code to the repository.

This part assumes you use a source control system to manage and store your code. Examples include Git, Mercurial, SVN, and several others. Git is a very popular source control.

A third level is the level of APIs where we encapsulate potential future changes to our code by only allowing interactions with our code via published interfaces.

Another level of testing is integration testing, when half of the bridge we built meets the other half that the other development teams created, and the two don't meet at the same height (say, one half ended up two meters or yards higher than the other half...).

Then, there is end user testing. While we built what they specified, it is not really what they wanted.

All of the preceding examples are valid reasons why we need to test our code both in the design and implementation stages.

Getting ready

We will test the GUI we created in recent recipes and chapters. We will also show some simple examples of what can go wrong and why we need to keep testing our code and the code we call via APIs.

How to do it...

Let's examine this recipe in detail.

In Python GUI programming, one of the first things that can go wrong is missing out on importing required modules.

Here is a simple example:

1. Open `GUI.py` and comment out the `import statement, # import tkinter as tk.`
2. Run the code and observe the following output:

The screenshot shows an IDE interface with a code editor and a terminal window. The code editor contains a Python class definition:

```
#=====
# imports
#=====
class OOP():
    def __init__(self, language='en'):
        # Create instance
        self.win = tk.Tk()
```

The terminal window shows the execution of the script and a resulting traceback:

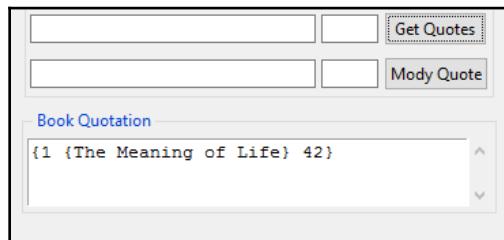
```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python
Traceback (most recent call last):
  File "C:\Eclipse_NEON_workspace\2nd_Edition_OOP.py", line 1, in <module>
    oop = OOP()
  File "C:\Eclipse_NEON_workspace\2nd_Edition_OOP.py", line 3, in __init__
    self.win = tk.Tk()
NameError: name 'tk' is not defined
```

3. Add the `import statement` to the top to solve this error as follows:

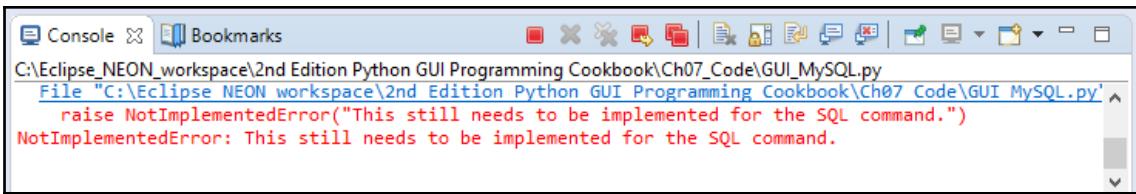
```
#=====
# imports
#=====
import tkinter as tk
```

Using an example from Chapter 7, *Storing Data in Our MySQL Database via Our GUI*, let's say we click on the **Get Quotes** button and this works, but we never clicked on the **Mody Quote** button.

4. Open `GUI_MySQL.py` from Chapter 7, *Storing Data in Our MySQL Database via Our GUI*, and click the **Get Quotes** button:

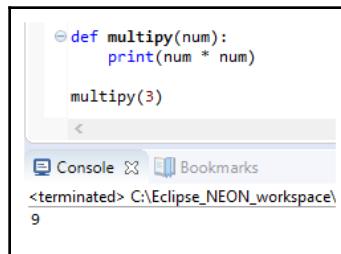


5. Next, click the **Mody Quote** button, which creates the following result:



The screenshot shows the Eclipse IDE interface with a focus on the Console view. The title bar indicates the workspace path: C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch07_Code\GUI MySQL.py. The console output window displays the following text:
File "C:/Eclipse_NEON_workspace/2nd_Edition_Python_GUI_Programming_Cookbook/Ch07_Code/GUI MySQL.py"
 raise NotImplementedError("This still needs to be implemented for the SQL command.")
NotImplementedError: This still needs to be implemented for the SQL command.

6. Another potential area of bugs is when a function or method suddenly no longer returns the expected result. Let's say we are calling the following function, which returns the expected result:



The screenshot shows the Eclipse IDE interface. The code editor contains the following Python code:

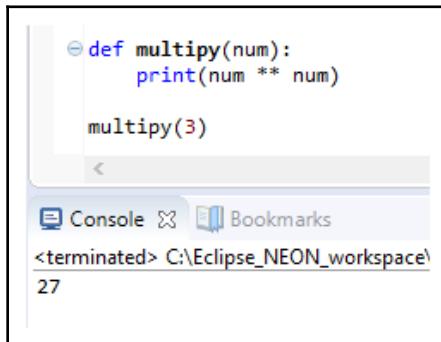
```
def multiply(num):
    print(num * num)

multiply(3)
```

Below the code editor is a terminal window titled 'Console' showing the output of the code execution:
<terminated> C:\Eclipse_NEON_workspace\
9

7. Then, someone makes a mistake, and we no longer get the previous results.

Change `(num * num)` to `(num ** num)` and run the code:



The screenshot shows the Eclipse IDE interface. The code editor contains the same Python code as before, but with a change in the multiplication operator:

```
def multiply(num):
    print(num ** num)

multiply(3)
```

Below the code editor is a terminal window titled 'Console' showing the output of the code execution:
<terminated> C:\Eclipse_NEON_workspace\
27

Let's now go behind the scenes to understand the code better.

How it works...

First, in *Steps 1* and *2*, we are trying to create an instance of the `tkinter` class, but things don't work as expected.

Well, we simply forgot to import the module and alias it as `tk`, and we can fix this by adding a line of Python code above our class creation, where the `import` statements live.

This is an example in which our development environment does the testing for us. We just have to do the debugging and code fixing.

Another example more closely related to developer testing is when we code conditionals and, during our regular development, do not exercise all branches of logic.

To demonstrate this, in *Steps 4* and *5*, we use an example from [Chapter 7, Storing Data in Our MySQL Database via Our GUI](#). We click on the **Get Quotes** button and this works, but we never clicked on the **Modify Quote** button. The first button click creates the desired result, but the second throws an exception (because we had not yet implemented this code and probably forgot all about it).

In the next example, in *Steps 6* and *7*, instead of multiplying, we are exponentially raising by the power of the passed-in number, and the result is no longer what it used to be.

In software testing, this sort of bug is called **regression**.



Whenever something goes wrong in our code, we have to debug it. The first step of doing this is to set breakpoints and then step through our code, line by line, or method by method.

Stepping in and out of our code is a daily activity until the code runs smoothly.

In this recipe, we emphasized the importance of software testing during several phases of the software development life cycle by showing several examples of where the code can go wrong and introduce software defects (also known as bugs).

Setting debug watches

In modern IDEs, such as the PyDev plugin in Eclipse, or other IDEs such as NetBeans, we can set debug watches to monitor the state of our GUI during the execution of our code.

This is very similar to the Microsoft IDEs of Visual Studio and the more recent versions of Visual Studio .NET.



Setting debug watches is a very convenient way to help our development efforts.

Getting ready

In this recipe, we will reuse the Python GUI we developed in the earlier recipes. We will step through the code we had developed previously, and we will set debug watches.

How to do it...

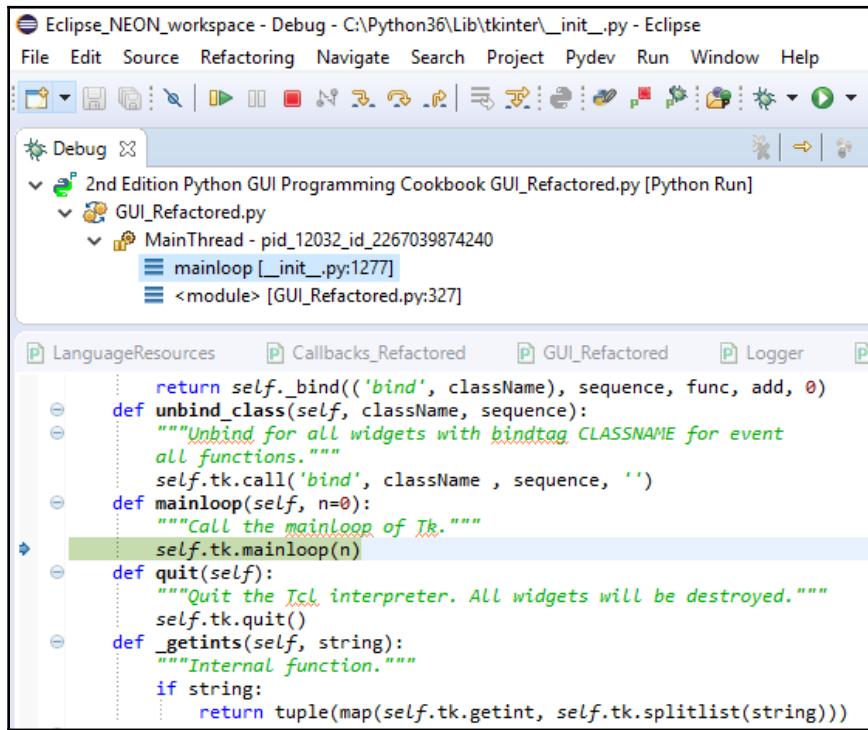
While this recipe applies to the PyDev plugin in the Java-based Eclipse IDE, its principles also apply to many modern IDEs.

Let's now see how we can sequentially proceed with this recipe:

1. Open `GUI.py` and place a breakpoint at the line with `mainloop`:

```
#=====
# Start GUI
=====
oop = OOP()
oop.win.mainloop()
```

2. Start a debug session as follows:



Let's place a breakpoint at the **New York** button callback method, which we named `getDateTime`.

3. Open `Callbacks_Refactored.py` and place a breakpoint at the `getDateTime` method.

4. Step through the code:

The screenshot shows the PyDev/Eclipse debugger interface during a step-through session. At the top, there are tabs for 'Variables', 'Breakpoints', and 'Expressions'. Below these, a table lists variables with their current values:

Name	Value
> <code>utc</code>	<code>datetime: 2016-12-19 02:18:34.697874+00:00</code>
> <code>la</code>	<code>datetime: 2016-12-18 18:18:34.697874-08:00</code>
> <code>nv</code>	<code>datetime: 2016-12-18 21:18:34.697874-05:00</code>

A message below the table states: "No details to display for the current selection." Below this, the code editor shows a Python script with several lines of code. The line `self.oop.lbl2.set(ny.strftime(fmtStrZone))` is highlighted in green, indicating it is the next line to be executed. The code itself includes logic for formatting UTC datetime objects into local US time zones (Los Angeles and New York) and updating a GUI label with the results.

```

self.oop.scr.delete('1.0', tk.END)
self.oop.scr.insert(tk.INSERT, get_localzone())

# Format local US time with TimeZone info
def getDateTime(self):
    fmtStrZone = "%Y-%m-%d %H:%M:%S %Z%z"
    # Get Coordinated Universal Time
    utc = datetime.now(timezone('UTC'))
    self.oop.log.writeToLog(utc.strftime(fmtStrZone),
                           self.oop.level.MINIMUM)

    # Convert UTC datetime object to Los Angeles TimeZone
    la = utc.astimezone(timezone('America/Los_Angeles'))
    self.oop.log.writeToLog(la.strftime(fmtStrZone),
                           self.oop.level.NORMAL)

    # Convert UTC datetime object to New York TimeZone
    ny = utc.astimezone(timezone('America/New_York'))
    self.oop.log.writeToLog(ny.strftime(fmtStrZone),
                           self.oop.level.DEBUG)

    # update GUI label with NY Time and Zone
    self.oop.lbl2.set(ny.strftime(fmtStrZone))

```

Let's now go behind the scenes to understand the code better.

How it works...

The first position where we might wish to place a breakpoint is at the place where we make our GUI visible by calling the `tkinter` main event loop. We do this in *step 1*.

The green balloon symbol on the left is a breakpoint in PyDev/Eclipse. When we execute our code in debug mode, the execution of the code will be halted once the execution reaches the breakpoint. At this point, we can see the values of all the variables that are currently in scope. We can also type expressions into one of the debugger windows, which will execute them, showing us the results. If the result is what we want, we might decide to change our code using what we have just learned.

We normally step through the code by either clicking an icon in the toolbar of our IDE, or by using a keyboard shortcut (such as pressing *F5* to step into code, *F6* to step over, and *F7* to step out of the current method).

Placing the breakpoint where we did and then stepping into this code turns out to be a problem because we end up in some low-level `tkinter` code we really do not wish to debug right now. We get out of this low-level `tkinter` code by clicking the Step-Out toolbar icon (which is the third yellow arrow on the right beneath the project menu) or by pressing *F7* (assuming we are using PyDev in Eclipse).

A better idea is to place our breakpoint closer to our own code in order to watch the values of some of our own Python variables. In the event-driven world of modern GUIs, we have to place our breakpoints at code that gets invoked during events, for example, button clicks. We do this in *Steps 3 and 4*.

Currently, one of our main functionalities resides in a button click event. When we click the button labelled **New York**, we create an event, which then results in something happening in our GUI.



If you are interested in learning how to install Eclipse with the PyDev plugin for Python, there is a great tutorial that will get you started with installing all the necessary free software and then introduce you to PyDev within Eclipse by creating a simple, working Python program: <http://www.vogella.com/tutorials/Python/article.html>.

We use modern IDEs in the 21st century that are freely available to help us to create solid code.

This recipe showed how to set debug watches, which is a fundamental tool in every developer's skill set. Stepping through our own code even when not hunting down bugs ensures that we understand our code, and it can lead to improving our code via refactoring.



Debug watches help us to create solid code and this is not a waste of time.

Configuring different debug output levels

In this recipe, we will configure different debug levels, which we can select and change at runtime. This allows us to control how much we want to drill down into our code when debugging our code.

We will create two new Python classes and place both of them in the same module.

We will use four different logging levels and write our debugging output to a log file that we will create. If the logs folder does not exist, we will create it automatically as well.

The name of the log file is the name of the executing script, which is our refactored `GUI_Refactored.py`. We can also choose other names for our log files by passing in the full path to the initializer of our `Logger` class.

Getting ready

We will continue using our refactored `GUI_Refactored.py` code from the previous recipe.

How to do it...

Let's see how we shall proceed with this recipe:

1. First, we create a new Python module into which we place two new classes. The first class is very simple and defines the logging levels. This is basically an enumeration.
2. Create a new module and name it `Logger.py`.
3. Add the code as follows:

```
class LogLevel:  
    '''Define logging levels.'''  
    OFF      = 0  
    MINIMUM = 1  
    NORMAL   = 2  
    DEBUG    = 3
```

4. Add the second class to the same module, `Logger.py`:

```
import os, time  
from datetime import datetime  
class Logger:  
    ''' Create a test log and write to it. '''  
    #-----  
    def __init__(self, fullTestName, loglevel=LogLevel.DEBUG):  
        testName =  
        os.path.splitext(os.path.basename(fullTestName))[0]  
        logName  = testName + '.log'  
  
        logsFolder = 'logs'
```

```
        if not os.path.exists(logsFolder):
            os.makedirs(logsFolder, exist_ok = True)

        self.log = os.path.join(logsFolder, logName)
        self.createLog()

        self.loggingLevel = loglevel
        self.startTime     = time.perf_counter()

#-----
def createLog(self):
    with open(self.log, mode='w', encoding='utf-8') as logFile:
        logFile.write(self.getDateTime() +
                      '\t\t*** Starting Test ***\n')
    logFile.close()
```

5. Add the `writeToLog` method shown here:

```
#-----
def writeToLog(self, msg='', loglevel=LogLevel.DEBUG):
    # control how much gets logged
    if loglevel > self.loggingLevel:
        return

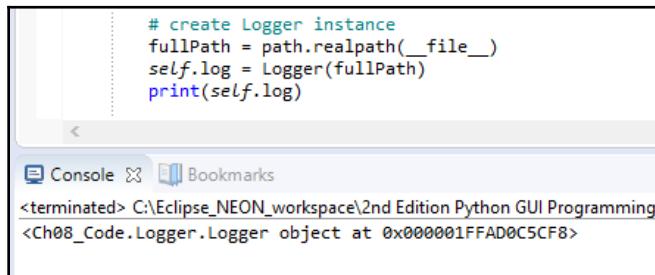
    # open log file in append mode
    with open(self.log, mode='a', encoding='utf-8') as logFile:
        msg = str(msg)
        if msg.startswith('\n'):
            msg = msg[1:]
        logFile.write(self.getDateTime() + '\t\t' + msg + '\n')

    logFile.close()
```

6. Open `GUI_Refactored.py` and add this code:

```
from os import path
from Ch08_Code.Logger import Logger
class OOP():
    def __init__(self):
        # create Logger instance
        fullPath = path.realpath(__file__)
        self.log = Logger(fullPath)
        print(self.log)
```

7. Run the code and observe the output:



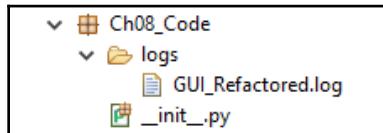
The screenshot shows a code editor window with the following Python code:

```
# create Logger instance
fullPath = path.realpath(__file__)
self.log = Logger(fullPath)
print(self.log)
```

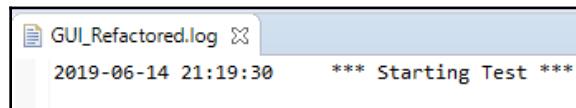
Below the code editor is a terminal window titled "Console". The output shows:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming
<Ch08_Code.Logger object at 0x000001FFAD0C5CF8>
```

The preceding screenshot shows that we created an instance of our new `Logger` class, and the following screenshot shows that both the `logs` folder as well as the log were created:



8. Finally, open the log file:



Let's now go behind the scenes to understand the code better.

How it works...

We first created a new module and used a simple class as an enumeration.

The second class creates a log file by using the passed-in full path of the filename and places this into a `logs` folder. On the first run, the `logs` folder might not exist, so the code automatically creates the folder.

In order to write to our log file, we use the `writeToLog` method. Inside the method, the first thing we do is check whether the message has a logging level higher than the limit we set our desired logging output to. If the message has a lower level, we discard it and immediately return from the method.

If the message has the logging level that we want to display, we then check whether it starts with a newline character, and, if it does, we discard the newline by slicing the method starting at index 1, using Python's slice operator (`msg = msg[1:]`).

We then write one line to our log file, consisting of the current date timestamp, two tab spaces, our message, and ending in a newline character.

We can now import our new Python module and, inside the `__init__` section of our GUI code, we create an instance of the `Logger` class.

We are retrieving the full path to our running GUI script via `path.realpath(__file__)` and passing this into the initializer of the `Logger` class. If the logs folder does not exist, it will automatically be created by our Python code.

We then verify that the log and folder got created.

In this recipe, we created our own logging class. While Python ships with a logging module, it is very easy to create our own, which gives us absolute control over our logging format. This is very useful when we combine our own logging output with MS Excel or Matplotlib, which we explored in the recipes of a previous chapter.

In the next recipe, we will use Python's built-in `__main__` functionality to use the four different logging levels we just created.

Creating self-testing code using Python's `__main__` section

Python comes with a very nice feature that enables each module to self-test. Making use of this feature is a great way of making sure that the changes to our code do not break the existing code and, additionally, the `__main__` self-testing section can serve as documentation for how each module works.

After a few months or years, we sometimes forget what our code is doing, so having an explanation written in the code itself is indeed of great benefit.

It is a good idea to always add a self-testing section to every Python module, when possible. It is sometimes not possible but, in most modules, it is possible to do so.

Getting ready

We will extend the previous recipe, so in order to understand what the code in this recipe is doing, we have to first read and understand the code of the previous recipe.

How to do it...

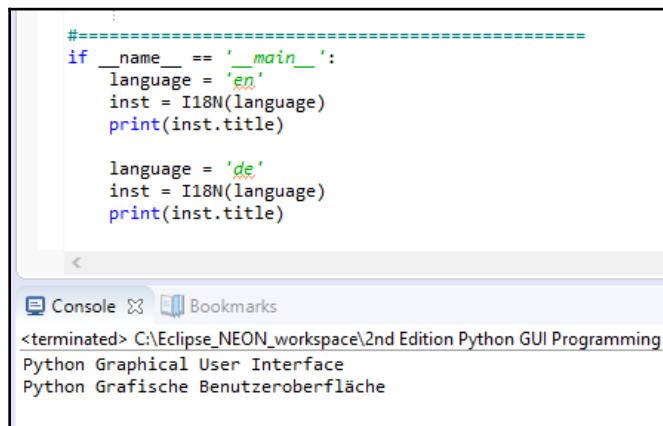
Let's see the steps of this recipe in detail:

1. First, we will explore the power of the Python `__main__` self-testing section by adding this self-testing section to our `LanguageResources.py` module.
2. Next, add the following code to `LanguageResources.py`:

```
if __name__ == '__main__':
    language = 'en'
    inst = I18N(language)
    print(inst.title)

    language = 'de'
    inst = I18N(language)
    print(inst.title)
```

3. Run the code and observe the output:

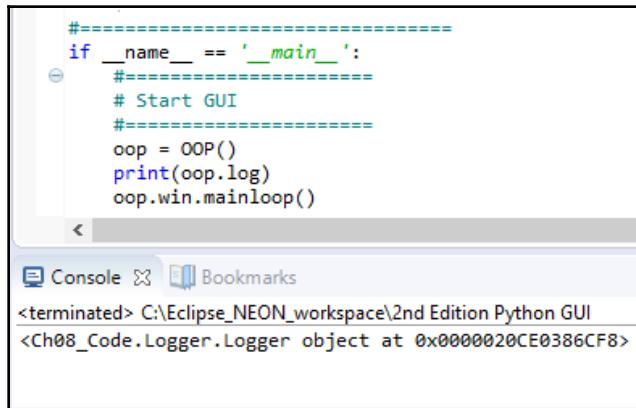


```
#=====
if __name__ == '__main__':
    language = 'en'
    inst = I18N(language)
    print(inst.title)

    language = 'de'
    inst = I18N(language)
    print(inst.title)
```

Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming
Python Graphical User Interface
Python Grafische Benutzeroberfläche

4. Add a `__main__` self-testing section to the `GUI_Refactored.py` module and run the code to see the following output:



The screenshot shows the Eclipse IDE interface. On the left is a code editor with Python code. On the right is a terminal window titled 'Console'.

```
#=====
if __name__ == '__main__':
#=====
# Start GUI
#=====
oop = OOP()
print(oop.log)
oop.win.mainloop()
```

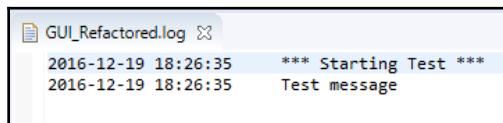
The terminal window displays the following output:

```
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI
<Ch08_Code.Logger.Logger object at 0x0000020CE0386CF8>
```

5. Next, in the `GUI_Refactored.py` module, add `oop.log.writeToLog('Test message')`:

```
if __name__ == '__main__':
#=====
# Start GUI
#=====
oop = OOP()
print(oop.log)
oop.log.writeToLog('Test message')
oop.win.mainloop()
```

This gets written to our log file, as can be seen in the following screenshot of the log:



The screenshot shows a log file titled 'GUI_Refactored.log'. It contains the following entries:

```
2016-12-19 18:26:35    *** Starting Test ***
2016-12-19 18:26:35    Test message
```

6. In `GUI_Refactored.py`, import both new classes from our `Logger` module:

```
from Ch08_Code.Logger import Logger, LogLevel
```

7. Next, create local instances of those classes:

```
# create Logger instance
fullPath = path.realpath(__file__)
self.log = Logger(fullPath)

# create Log Level instance
self.level = LogLevel()
```

8. Use different logging levels via self.oop.level:

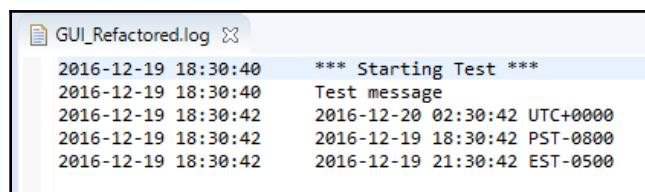
```
# Format local US time with TimeZone info
def getDateTime(self):
    fmtStrZone = "%Y-%m-%d %H:%M:%S %Z%z"
    # Get Coordinated Universal Time
    utc = datetime.now(timezone('UTC'))
    self.oop.log.writeToLog(utc.strftime(fmtStrZone),
                           self.oop.level.MINIMUM)

    # Convert UTC datetime object to Los Angeles TimeZone
    la = utc.astimezone(timezone('America/Los_Angeles'))
    self.oop.log.writeToLog(la.strftime(fmtStrZone),
                           self.oop.level.NORMAL)

    # Convert UTC datetime object to New York TimeZone
    ny = utc.astimezone(timezone('America/New_York'))
    self.oop.log.writeToLog(ny.strftime(fmtStrZone),
                           self.oop.level.DEBUG)

    # update GUI label with NY Time and Zone
    self.oop.lbl2.set(ny.strftime(fmtStrZone))
```

9. Run the code and open the log:



The screenshot shows a terminal window titled "GUI_Refactored.log". The log file contains the following entries:

Date	Time	Message
2016-12-19	18:30:40	*** Starting Test ***
2016-12-19	18:30:40	Test message
2016-12-19	18:30:42	2016-12-20 02:30:42 UTC+0000
2016-12-19	18:30:42	2016-12-19 18:30:42 PST-0800
2016-12-19	18:30:42	2016-12-19 21:30:42 EST-0500

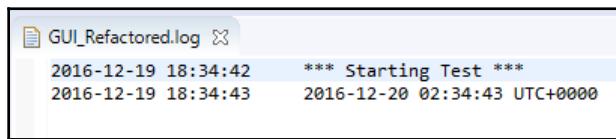
Notice the `setLoggingLevel` method of the `Logger` class:

```
#-----
def setLoggingLevel(self, level):
    '''change logging level in the middle of a test.'''
    self.loggingLevel = level
```

10. In the `__main__` section of our GUI, change the logging level to MINIMUM:

```
if __name__ == '__main__':
=====
# Start GUI
=====
oop = OOP()
oop.log.setLevel(oop.level.MINIMUM)
oop.log.writeToLog('Test message')
oop.win.mainloop()
```

11. Open the log file:



Let's now go behind the scenes to understand the code better.

How it works...

We start by adding a `__main__` self-testing section to `LanguageResources.py`.

Whenever we run a module that has this self-testing section located at the bottom of the module, when the module is executed by itself, this code will run.

When the module is imported and used from other modules, the code in the `__main__` self-testing section will not be executed.

We first pass English as the language to be displayed in our GUI, and then we pass German as the language our GUI will display.

We print out the title of our GUI to show that our Python module works as we intended it to work.

The next step is to use our logging capabilities, which we created in the previous recipe.

We add a `__main__` self-testing section to `GUI_Refactored.py` and then verify that we created an instance of our `Logger` class.

We next write to our log file by using the command shown. We have designed our logging level to default to log every message, which is the `DEBUG` level, and, because of this, we do not have to change anything. We just pass the message to be logged to the `writeToLog` method.

Now, we can control the logging by adding logging levels to our logging statements and setting the level we wish to output. We add this capability to our **New York** button callback method in the `Callbacks_Refactored.py` module, which is the `getDateTime` method.

We change the previous print statements to log statements using different debug levels.

As we are passing an instance of the `GUI` class to the `Callbacks_Refactored.py` initializer, we can use logging level constraints according to the `LogLevel` class we created.

When we now click our **New York** button, depending upon the logging level selected, we get different output written to our log file. The default logging level is `DEBUG`, which means that everything gets written to our log.

When we change the logging level, we control what gets written to our log. We do this by calling the `setLoggingLevel` method of the `Logger` class.

Setting the level to `MINIMUM` results in a reduced output written to our log file.

Now, our log file no longer shows the test message and only shows messages that meet the set logging level.

In this recipe, we made good use of Python's built-in `__main__` self-testing section. We introduced our own logging file and, at the same time, learned how to create different logging levels. By doing this, we have full control over what gets written to our log files.

Creating robust GUIs using unit tests

Python comes with a built-in unit testing framework and, in this recipe, we will start using this framework to test our Python GUI code.

Before we start writing unit tests, we want to design our testing strategy. We could easily intermix the unit tests with the code they are testing, but a better strategy is to separate the application code from the unit test code.



PyUnit has been designed according to the principles of all the other xUnit testing frameworks.

Getting ready

We will test the internationalized GUI we created earlier in this chapter.

How to do it...

In order to use Python's built-in unit testing framework, we import the Python `unittest` module. Let's now look at the next steps:

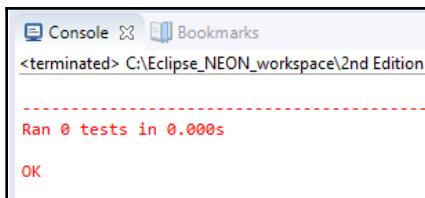
1. Create a new module and name it `UnitTestsMinimum.py`.
2. Add the following code:

```
import unittest

class GuiUnitTests(unittest.TestCase):
    pass

if __name__ == '__main__':
    unittest.main()
```

3. Run `UnitTestsMinimum.py` and observe the following output:



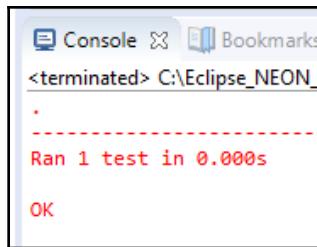
4. Create a new module, name it `UnitTests_One.py`, and then add this code:

```
import unittest
from Ch08_Code.LanguageResources import I18N

class GuiUnitTests(unittest.TestCase):

    def test_TitleIsEnglish(self):
        i18n = I18N('en')
        self.assertEqual(i18n.title,
                        "Python Graphical User Interface")
```

5. Run `UnitTests_One.py`:



6. Save the module as `UnitTestsFail.py` and then copy, paste, and modify the code:

```
import unittest
from Ch08_Code.LanguageResources import I18N

class GuiUnitTests(unittest.TestCase):

    def test_TitleIsEnglish(self):
        i18n = I18N('en')
        self.assertEqual(i18n.title, "Python Graphical User
Interface")

    def test_TitleIsGerman(self):
        i18n = I18N('en')
        self.assertEqual(i18n.title,
                        'Python Grafische Benutzeroberfl' + "u00E4" + 'che')
```

7. Run UnitTestsFail.py:



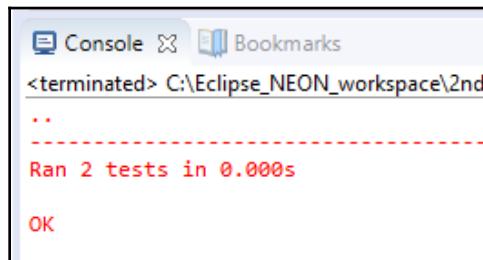
```
Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch08_Code\UnitTestsFail.py
.F
=====
FAIL: test_TitleIsGerman (__main__.GuiUnitTests)
-----
Traceback (most recent call last):
  File "C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch08_Code\UnitTestsFail.py", line 23, in test_TitleIsGerman
    + "\u00e4" + 'che')
AssertionError: 'Python Graphical User Interface' != 'Python Grafische Benutzeroberfl\u00e4che'
- Python Graphical User Interface
+ Python Grafische Benutzeroberfl\u00e4che

-----
Ran 2 tests in 0.001s
FAILED (failures=1)
```

8. Correct this failure by passing in 'de' to I18N:

```
def test_TitleIsGerman(self):
    i18n = I18N('en')                      # <= Bug in Unit Test
    i18n = I18N('de')
    self.assertEqual(i18n.title, 'Python Grafische Benutzeroberfl'
+ "\u00e4" + 'che')
```

9. Rerun UnitTestsFail.py with the failure corrected and observe the output:



```
Console Bookmarks
<terminated> C:\Eclipse_NEON_workspace\2nd
...
-----
Ran 2 tests in 0.000s
OK
```

Let's now go behind the scenes to understand the code better.

How it works...

We first import the `unittest` module, then we create our own class and, within this class, we inherit and extend the `unittest.TestCase` class. We use the minimum amount of code to get started. The code isn't doing much yet but, when we run it, we do not get any errors, which is a good sign.

We actually do get an output written to the console stating that we successfully ran zero tests.

That output is a bit misleading, as all we have done so far is create a class that contains no actual testing methods.

We add testing methods that do the actual unit testing by following the default naming for all the test methods to start with the word **test**. This is an option that can be changed, but it is much easier and clearer to follow this *naming convention*.

We then add one test method that tests the title of our GUI. This will verify that, by passing the expected arguments, we get the expected result.

We are importing our `I18N` class from our `LanguageResources.py` module, passing English as the language to be displayed in our GUI. As this is our first unit test, we will print out the title result as well, just to make sure we know what we are getting back. We next use the `unittest assertEquals` method to verify that our title is correct.

Running this code gives us an OK, which means that the unit test passed.

The unit test runs and succeeds, which is indicated by one dot and the word OK. If it had failed or got an error, we would not have gotten the dot but an F or E as the output.

We then do the same automated unit testing check by verifying the title for the German version of our GUI. We test our internationalized GUI title in two languages.

We ran two unit tests but, instead of an OK, we got a failure. What happened?

Our assertion failed for the German version of our GUI.

While debugging our code, it turns out that in the copy, paste, and modify approach of our unit test code, we forgot to pass in German as the language.

After correcting the failure, we reran our unit tests, and we get the expected result of all our tests passing.



Unit testing code is also code and can have bugs too.

While the purpose of writing unit tests is really to test our application code, we have to make sure that our tests are written correctly. One approach from the **Test-Driven-Development (TDD)** methodology might help us.



In TDD, we develop the unit tests before we actually write the application code. Now, if a test passes for a method that does not even exist, something is wrong. The next step is to create the non-existing method and make sure it will fail. After that, we can write the minimum amount of code necessary to make the unit test pass.

In this recipe, we started testing our Python GUI, and writing unit tests in Python. We saw that Python unit test code is just code and can contain mistakes that need to be corrected. In the next recipe, we will extend this recipe's code and use the graphical unit test runner that comes with the PyDev plugin for the Eclipse IDE.

How to write unit tests using the Eclipse PyDev IDE

In the previous recipe, we started using Python's unit testing capabilities, and, in this recipe, we will ensure the quality of our GUI code by using this capability further.

We will unit test our GUI in order to make sure that the internationalized strings our GUI displays are as expected.

In the previous recipe, we encountered some bugs in our unit testing code but, typically, our unit tests will find regression bugs that are caused by modifying the existing application code, not the unit test code. Once we have verified that our unit testing code is correct, we do not usually change it.



Our unit tests also serve as a documentation of what we expect our code to do.

By default, Python's unit tests are executed with a textual unit test runner, and we can run this in the PyDev plugin from within the Eclipse IDE. We can also run the very same unit tests from a console window.

In addition to the text runner in this recipe, we will explore PyDev's graphical unit test feature, which can be used from within the Eclipse IDE.

Getting ready

We will extend the previous recipe in which we began using Python unit tests. The Python unit testing framework comes with what are called **Test Fixtures**.

Refer to the following URLs for a description of what a test fixture is:

- <https://docs.python.org/3.7/library/unittest.html>
- https://en.wikipedia.org/wiki/Test_fixture
- http://www.boost.org/doc/libs/1_51_0/libs/test/doc/html/utf/user-guide.fixture.html

What this means is that we can create `setup` and `teardown` unit testing methods so that the `setup` method is called at the beginning before any single test is executed, and, at the end of every single unit test, the `teardown` method is called.



This test fixture capability provides us with a very controlled environment in which we can run our unit tests.

How to do it...

Now, let's see how to perform this recipe:

1. Firstly, let's set up our unit testing environment. We will create a new testing class that focuses on the aforementioned correctness of code.
2. Create a new module, `UnitTestsEnglish.py`.
3. Add the following code:

```
import unittest
from Ch08_Code.LanguageResources import I18N
from Ch08_Code.GUI_Refactored import OOP as GUI

class GuiUnitTests(unittest.TestCase):
    def test_TitleIsEnglish(self):
        i18n = I18N('en')
        self.assertEqual(i18n.title,
                        "Python Graphical User Interface")

    def test_TitleIsGerman(self):
        # i18n = I18N('en') # <= Bug in Unit Test
        i18n = I18N('de')
```

```
        self.assertEqual(i18n.title,
                      'Python Grafische Benutzeroberfl' + "u00E4" + 'che')

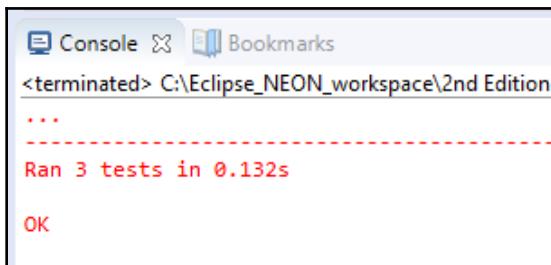
    class WidgetsTestsEnglish(unittest.TestCase):
        def setUp(self):
            self.gui = GUI('en')

        def tearDown(self):
            self.gui = None

        def test_WidgetLabels(self):
            self.assertEqual(self.gui.i18n.file, "File")
            self.assertEqual(self.gui.i18n.mgrFiles, ' Manage Files ')
            self.assertEqual(self.gui.i18n.browseTo,
                            "Browse to File...")

#=====
if __name__ == '__main__':
    unittest.main()
```

4. Run the code and observe the output:



5. Open `UnitTestsEnglish.py` and save it as `UnitTests.py`.
6. Add the following code to the module:

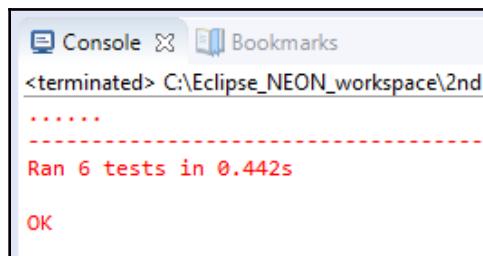
```
class WidgetsTestsGerman(unittest.TestCase):
    def setUp(self):
        self.gui = GUI('de')

    def test_WidgetLabels(self):
        self.assertEqual(self.gui.i18n.file, "Datei")
        self.assertEqual(self.gui.i18n.mgrFiles, ' Dateien
Organisieren ')
        self.assertEqual(self.gui.i18n.browseTo, "Waehle eine
Datei... ")

    def test_LabelFrameText(self):
```

```
labelFrameText = self.gui.widgetFrame['text']
self.assertEqual(labelFrameText, " Widgets Rahmen ")
self.gui.radVar.set(1)
self.gui.callBacks.radCall()
labelFrameText = self.gui.widgetFrame['text']
self.assertEqual(labelFrameText, " Widgets Rahmen in Gold")
  
...
```

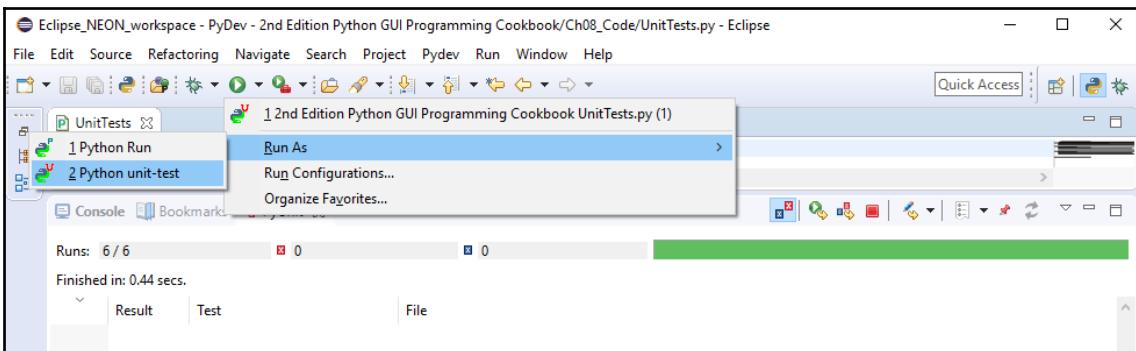
7. Run UnitTests.py:



8. Run the code from Command Prompt and observe the output as follows:

```
C:\Eclipse_Oxygen_workspace_Packt_3rd_GUI_BOOK\3rd Edition Python GUI Programming Cookbook\Ch08_Code>python UnitTests.py
.....
Ran 6 tests in 0.562s
OK
```

9. Using Eclipse, we can also choose to run our unit tests, not as a simple Python script, but as a Python unit test script, which gives us some colorful output:



Let's learn about the steps in the next section.

How it works...

Firstly, we created three test methods.



`unittest.main()` runs any method that starts with the `test` prefix, no matter how many classes we create within a given Python module.

The unit testing code shows that we can create several unit testing classes and they can all be run in the same module by calling `unittest.main()`.

It also shows that the `setup` method does not count as a test in the output of the unit test report (the count of tests is three) while, at the same time, it did its intended job as we can now access our `self.gui` class instance variable from within the unit test method.

We are interested in testing the correctness of all of our labels and, especially, catching bugs when we make changes to our code.

If we have copied and pasted strings from our application code to the testing code, it will catch any unintended changes with the click of a unit testing framework button.

We also want to test the fact that invoking any of our radio button widgets in any language results in the `LabelFrame` widget text being updated. In order to automatically test this, we have to do two things.

First, we have to retrieve the value of the `LabelFrame` widget and assign the value to a variable we name `labelFrameText`. We have to use the following syntax because the properties of this widget are being passed and retrieved via a dictionary data type:

```
self.gui.widgetFrame['text']
```

We can now verify the default text and then the internationalized versions after clicking one of the radio button widgets programmatically.

After verifying the default `labelFrameText`, we programmatically set the radio button to `index 1` and then invoke the radio button's callback method:

```
self.gui.radVar.set(1)  
self.gui.callBacks.radCall()
```



This is basically the same action as clicking the radio button in the GUI, but we do this button click event via code in the unit tests.



Then, we verify that our text in the `LabelFrame` widget has changed as intended.

If you get a `ModuleNotFoundError`, simply add the directory where your Python code lives to the Windows `PYTHONPATH` environmental variable, as shown in the following screenshots:

An error gets encountered as shown:

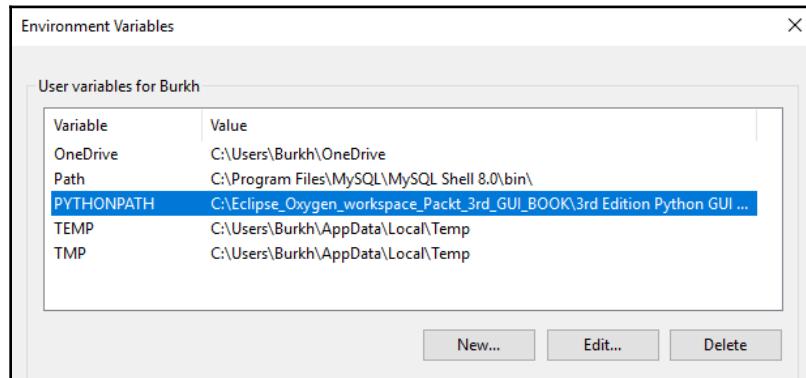
```
cmd Command Prompt
Microsoft Windows [Version 10.0.17134.829]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Burkh>cd C:\Eclipse_Oxygen_workspace_Packt_3rd_GUI_BOOK\3rd Edition Python GUI Programming Cookbook\Ch08_Code
C:\Eclipse_Oxygen_workspace_Packt_3rd_GUI_BOOK\3rd Edition Python GUI Programming Cookbook\Ch08_Code>python UnitTests.py

Traceback (most recent call last):
  File "UnitTests.py", line 8, in <module>
    from Ch08_Code.LanguageResources import I18N
ModuleNotFoundError: No module named 'Ch08_Code'

C:\Eclipse_Oxygen_workspace_Packt_3rd_GUI_BOOK\3rd Edition Python GUI Programming Cookbook\Ch08_Code>
```

The solution to the error, if you encounter it, is shown here:



For example, C:\Eclipse_Oxygen_workspace_Packt_3rd_GUI_BOOK\3rd Edition Python GUI Programming Cookbook:

OS (C:) > Eclipse_Oxygen_workspace_Packt_3rd_GUI_BOOK > 3rd Edition Python GUI Programming Cookbook			
Name	Date modified	Type	Size
.settings	5/29/2019 2:48 AM	File folder	
Ch01_Code	4/30/2019 2:07 PM	File folder	
Ch02_Code	5/15/2019 7:23 PM	File folder	
Ch03_Code	5/3/2019 12:59 AM	File folder	
Ch04_Code	6/14/2019 12:59 PM	File folder	
Ch05_Code	5/24/2019 12:19 PM	File folder	
Ch06_Code	6/14/2019 1:01 PM	File folder	
Ch07_Code	6/14/2019 1:04 PM	File folder	
Ch08_Code	6/14/2019 1:09 PM	File folder	

This will recognize the Ch08_Code folder as a Python package and the code will run.

When we run the unit test from the graphical runner in Eclipse, the result bar is green, which means that all our unit tests have passed.

We extended our unit testing code by testing labels, programmatically invoking Radiobutton, and then verifying in our unit tests that the corresponding text property of the LabelFrame widget has changed as expected. We tested two different languages. We then moved on to use the built-in Eclipse/PyDev graphical unit test runner.

9

Extending Our GUI with the wxPython Library

In this chapter, we will introduce another Python GUI toolkit that does not ship with Python. It is called wxPython. There are two versions of this library. The original is called **Classic**, while the newest is called by its development project code name, which is **Phoenix**.



The older Classic version does not work with Python 3.x, and we will not look further into this version but instead concentrate on the Phoenix software version.

In this book, we are solely programming using Python 3.7 and later, and because the new Phoenix project is also aimed at supporting Python 3.7 and later, this is the version of wxPython we will use in this chapter.

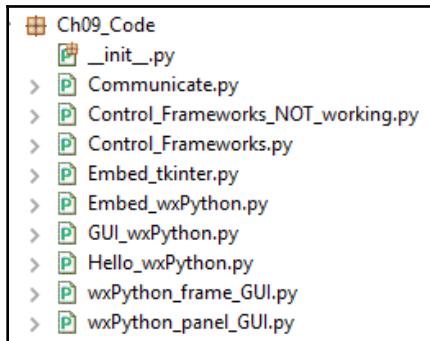
First, we will install the framework. Then, we will create a simple wxPython GUI, and after that, we will try to connect the `tkinter`-based GUIs we developed in this book with the new wxPython library.



wxPython is a Python binding to wxWidgets. The *w* in wxPython stands for the Windows OS, and the *x* stands for Unix-based OSes, such as Linux and Apple's macOS.

While `tkinter` ships with Python, it is valuable to have experience using other GUI frameworks that work with Python. This will improve your Python GUI programming skills, and you can choose which framework to use in your projects.

Here is the overview of Python modules for this chapter:



In this chapter, we will enhance our Python GUI by using the wxPython library. We will cover the following recipes:

- Installing the wxPython library
- Creating our GUI in wxPython
- Quickly adding controls using wxPython
- Trying to embed a main wxPython app in a main tkinter app
- Trying to embed our tkinter GUI code into wxPython
- Using Python to control two different GUI frameworks
- Communicating between two connected GUIs

Installing the wxPython library

The wxPython library does not ship with Python, so in order to use it, we first have to install it. This recipe will show us where and how to find the right version to install in order to match both the installed version of Python and the OS we are running.



The wxPython third-party library has been around for more than 18 years, which indicates that it is a robust library.

Getting ready

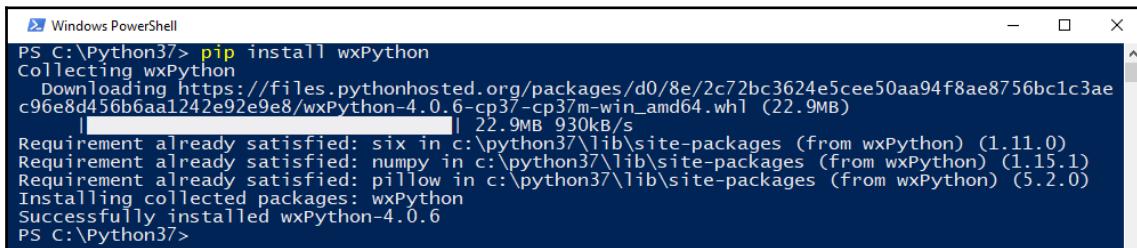
In order to use wxPython with Python 3.7 and later, we have to install the wxPython Phoenix version. Here is a link to the downloads page: <https://wxpython.org/pages/downloads/>. We will use this link to download and install the wxPython GUI framework.

And this is a link to PyPI with good information about how to use wxPython: <https://pypi.org/project/wxPython/>.

How to do it...

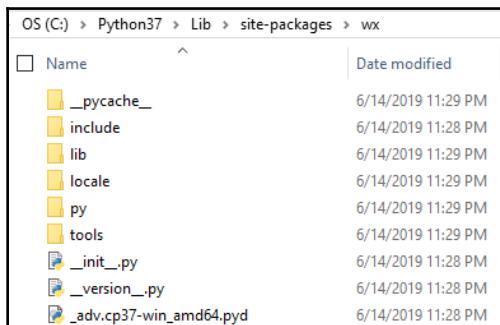
While only a few years ago it was a little tricky to find the right wxPython version for Python 3, we can now simply use pip. Let's see how in detail:

1. Open a Command Prompt or PowerShell window.
2. Type `pip install wxPython`.
3. The result should look similar to this:



```
PS C:\Python37> pip install wxPython
Collecting wxPython
  Downloading https://files.pythonhosted.org/packages/d0/8e/2c72bc3624e5cee50aa94f8ae8756bc1c3ae
c96e8d456b6aa1242e92e9e8/wxPython-4.0.6-cp37-cp37m-win_amd64.whl (22.9MB)
    |████████████████████████████████| 22.9MB 930kB/s
Requirement already satisfied: six in c:\python37\lib\site-packages (from wxPython) (1.11.0)
Requirement already satisfied: numpy in c:\python37\lib\site-packages (from wxPython) (1.15.1)
Requirement already satisfied: pillow in c:\python37\lib\site-packages (from wxPython) (5.2.0)
Installing collected packages: wxPython
Successfully installed wxPython-4.0.6
PS C:\Python37>
```

4. Verify that you have a new folder named `wx` in your Python `site-packages` folder:



5. Create a new module and call it `Hello_wxPython.py`.
6. Add the following code:

```
import wx
app = wx.App()
frame = wx.Frame(None, -1, "Hello World")
frame.Show()
app.MainLoop()
```

7. Running the preceding Python 3.7 script creates the following GUI using wxPython/Phoenix:



Now, let's go behind the scenes to understand the code better.

How it works...

First, we use `pip` to install the wxPython framework. Then, we verify that we have the new `wx` folder.



`wx` is the name of the folder that the wxPython Phoenix library was installed into. We will import this module into our Python code using the name `wx`.

We can verify that our installation worked by executing this simple demo script from the official wxPython/Phoenix website. The link to the official website is <https://wxpython.org/pages/overview/#hello-world>.

In this recipe, we successfully installed the correct version of the wxPython toolkit that we can use with Python 3.7. We found the Phoenix project for this GUI toolkit, which is the current and active development line. Phoenix will replace the classic wxPython toolkit in time and is specifically designed to work well with Python 3.7.

After successfully installing the wxPython/Phoenix toolkit, we then created a GUI using this toolkit in only five lines of code.



We previously achieved the same results by using `tkinter`.

Creating our GUI in *wxPython*

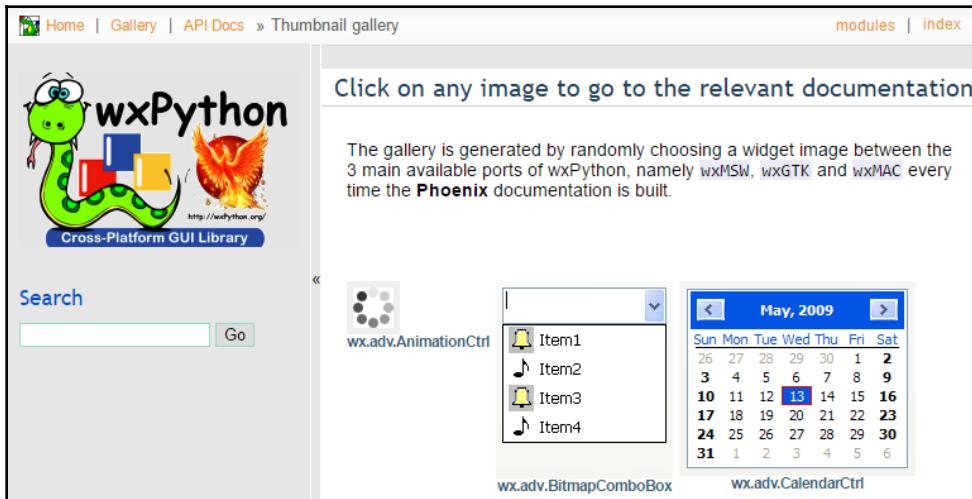
In this recipe, we will start creating our Python GUIs using the *wxPython* GUI toolkit. We will first recreate several of the widgets we previously created using `tkinter`, which ships with Python. Then, we will explore some of the widgets the *wxPython* GUI toolkit offers, which are not that easy to create by using `tkinter`.

Getting ready

The previous recipe showed you how to install the correct version of *wxPython* that matches both your version of Python and the OS you are running. A good place to start exploring the *wxPython* GUI toolkit is by going to the following URL:

<http://wxpython.org/Phoenix/docs/html/gallery.html>.

This web page displays many *wxPython* widgets and, by clicking on any of them, we are taken to their documentation, which is a very helpful feature if you want to quickly learn about a *wxPython* control:



Let's now use the wxPython library.

How to do it...

We can very quickly create a working window that comes with a title, a menu bar, and also a status bar. This status bar displays the text of a menu item when hovering the mouse over it. Moving forward, perform the following steps:

1. Create a new Python module and name it `wxPython_frame_GUI.py`.
2. Add the following code:

```
# Import wxPython GUI toolkit
import wx

# Subclass wxPython frame
class GUI(wx.Frame):
    def __init__(self, parent, title, size=(200,100)):
        # Initialize super class
        wx.Frame.__init__(self, parent, title=title, size=size)

        # Change the frame background color
        self.SetBackgroundColour('white')

        # Create Status Bar
        self.CreateStatusBar()

        # Create the Menu
        menu= wx.Menu()

        # Add Menu Items to the Menu
        menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
        menu.AppendSeparator()
        menu.Append(wx.ID_EXIT, "Exit", " Exit the GUI")

        # Create the MenuBar
        menuBar = wx.MenuBar()

        # Give the Menu a Title
        menuBar.Append(menu, "File")

        # Connect the MenuBar to the frame
        self.SetMenuBar(menuBar)

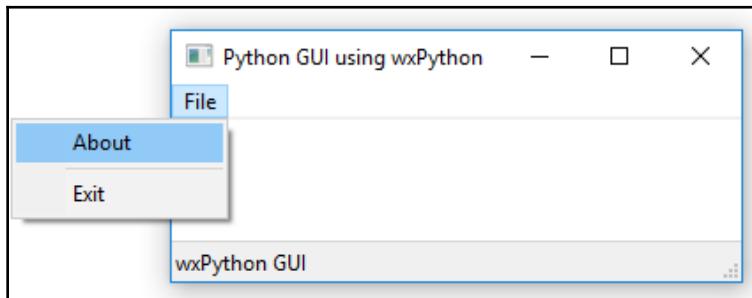
        # Display the frame
        self.Show()
```

```
# Create instance of wxPython application
app = wx.App()

# Call sub-classed wxPython GUI increasing default Window size
GUI(None, "Python GUI using wxPython", (300,150))

# Run the main GUI event loop
app.MainLoop()
```

3. This creates the following GUI, which is written in Python using the wxPython library:



4. Create a new module and name it `wxPython_panel_GUI.py`.
5. Add the following code:

```
import wx                                         # Import wxPython GUI
toolkit
class GUI(wx.Panel):                            # Subclass wxPython
    Panel
    def __init__(self, parent):
        # Initialize super class
        wx.Panel.__init__(self, parent)

        # Create Status Bar
        parent.CreateStatusBar()

        # Create the Menu
        menu= wx.Menu()

        # Add Menu Items to the Menu
        menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
        menu.AppendSeparator()
        menu.Append(wx.ID_EXIT, "Exit", " Exit the GUI")

        # Create the MenuBar
```

```
menuBar = wx.MenuBar()

# Give the Menu a Title
menuBar.Append(menu, "File")

# Connect the MenuBar to the frame
parent.SetMenuBar(menuBar)

# Create a Print Button
button = wx.Button(self, label="Print", pos=(0,60))

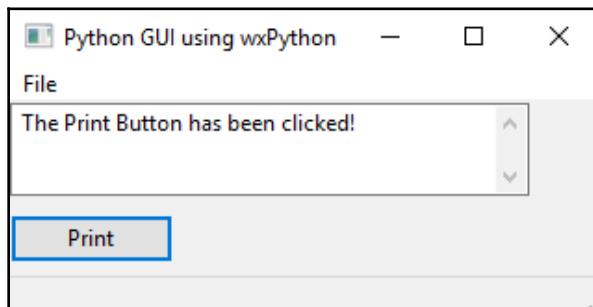
# Connect Button to Click Event method
self.Bind(wx.EVT_BUTTON, self.printButton, button)

# Create a Text Control widget
self.textBox = wx.TextCtrl(self, size=(280,50),
style=wx.TE_MULTILINE)

# callback event handler
def printButton(self, event):
    self.textBox.AppendText(
        "The Print Button has been clicked!")

app = wx.App()          # Create instance of wxPython application
                        # Create frame
frame = wx.Frame(None, title="Python GUI using wxPython",
size=(300,180))
GUI(frame)              # Pass frame into GUI
frame.Show()             # Display the frame
app.MainLoop()           # Run the main GUI event loop
```

7. Running the preceding code and clicking our wxPython button widget results in the following GUI output:



Now, let's go behind the scenes to understand the code better.

How it works...

In the `wxPython_frame_GUI.py` code, we inherit from `wx.Frame`. In the next code, we inherit from `wx.Panel` and we pass in `wx.Frame` to the `__init__()` method of our class.



In wxPython, the top-level GUI window is called a frame. There cannot be a wxPython GUI without a frame, and the frame has to be created as part of a wxPython application. We create both the application and the frame at the bottom of our code.

In order to add widgets to our GUI, we have to attach them to a panel. The parent of the panel is the frame (our top-level window), and the parent of the widgets we place into the panel is the panel. In the `wxPython_panel_GUI.py` code, the parent is a `wx.Frame` we are passing into the GUI initializer. We also add a button widget to the panel widget, which, when clicked, prints out some text to the textbox.

We have created our own GUI in this recipe using the mature wxPython GUI toolkit. In only a few lines of Python code, we were able to create a fully functional GUI that comes with minimize, maximize, and exit buttons. We added a menu bar, a multi-line text control, and a button. We also created a status bar that displays text when we select a menu item. We placed all these widgets into a panel container widget. We hooked up the button to print to the text control. When hovering over a menu item, some text gets displayed in the status bar.

Quickly adding controls using wxPython

In this recipe, we will recreate the GUI we originally created earlier in this book with `tkinter`, but this time, we will be using the wxPython library. We will see how easy and quick it is to use the wxPython GUI toolkit to create our own Python GUIs.

We will not recreate the entire functionality we created in the previous chapters. For example, we will not internationalize our wxPython GUI, nor connect it to a MySQL database. We will recreate the visual aspects of the GUI and add some functionality.



Comparing different libraries gives us the choice of which toolkits to use for our own Python GUI development, and we can combine several of those toolkits in our own Python code.

Getting ready

Ensure you have the wxPython module installed to follow this recipe.

How to do it...

Let's see how to perform this recipe:

1. First, we create our Python OOP class as we did before when using `tkinter`, but this time we inherit from and extend the `wx.Frame` class. We name the class `MainFrame`.
2. Create a new Python module and call it `GUI_wxPython.py`.
3. Add the following code:

```
import wx
BACKGROUNDCOLOR = (240, 240, 240, 255)

class MainFrame(wx.Frame):
    def __init__(self, *args, **kwargs):
        wx.Frame.__init__(self, *args, **kwargs)
        self.createWidgets()
        self.Show()

    def exitGUI(self, event):          # callback
        self.Destroy()

    def createWidgets(self):
        self.CreateStatusBar()         # wxPython built-in method
        self.createMenu()
        self.createNotebook()
```

4. Add the following code to create a notebook widget:

```
#-----
def createNotebook(self):
    panel = wx.Panel(self)
    notebook = wx.Notebook(panel)
    widgets = Widgets(notebook) # Custom class explained below
```

```
notebook.AddPage(widgets, "Widgets")
notebook.SetBackgroundColour(BACKGROUNDCOLOR)
# layout
boxSizer = wx.BoxSizer()
boxSizer.Add(notebook, 1, wx.EXPAND)
panel.SetSizerAndFit(boxSizer)
```

5. Add a new class and name it Widgets:

```
class Widgets(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent)
        self.createWidgetsFrame()
        self.addWidget()
        self.layoutWidgets()
```

6. Add these methods:

```
#-----
def createWidgetsFrame(self):
    self.panel = wx.Panel(self)
    staticBox = wx.StaticBox(self.panel, -1, "Widgets Frame")
    self.statBoxSizerV = wx.StaticBoxSizer(staticBox, wx.VERTICAL)
#-----
def layoutWidgets(self):
    boxSizerV = wx.BoxSizer(wx.VERTICAL)
    boxSizerV.Add(self.statBoxSizerV, 1, wx.ALL)
    self.panel.SetSizer(boxSizerV)
    boxSizerV.SetSizeHints(self.panel)
#-----
def addWidgets(self):
    self.addCheckboxes()
    self.addRadioButtons()
    self.addStaticBoxWithLabels()
```

7. Add the addStaticBoxWithLabels method:

```
def addStaticBoxWithLabels(self):
    boxSizerH = wx.BoxSizer(wx.HORIZONTAL)
    staticBox = wx.StaticBox(self.panel, -1, "Labels within a Frame")
    staticBoxSizerV = wx.StaticBoxSizer(staticBox, wx.VERTICAL)
    boxSizerV = wx.BoxSizer( wx.VERTICAL )
    staticText1 = wx.StaticText(self.panel, -1, "Choose a number:")
    boxSizerV.Add(staticText1, 0, wx.ALL)
    staticText2 = wx.StaticText(self.panel, -1,"Label 2")
    boxSizerV.Add(staticText2, 0, wx.ALL)
    #-----
    staticBoxSizerV.Add(boxSizerV, 0, wx.ALL)
```

```
boxSizerH.Add(staticBoxSizerV)
#-----
boxSizerH.Add(wx.TextCtrl(self.panel))
# Add local boxSizer to main frame
self.statBoxSizerV.Add(boxSizerH, 1, wx.ALL)
```

8. Add the following methods and call them in `__init__`:

```
class Widgets(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent)
        self.panel = wx.Panel(self)
        self.createWidgetsFrame()
        self.createManageFilesFrame()
        self.addWidget()
        self.addFileWidgets()
        self.layoutWidgets()
    #-----
    def createWidgetsFrame(self):
        staticBox = wx.StaticBox(self.panel, -1, "Widgets Frame",
                               size=(285, -1))
        self.statBoxSizerV = wx.StaticBoxSizer(staticBox,
                                              wx.VERTICAL)
    #-----
    def createManageFilesFrame(self):
        staticBox = wx.StaticBox(self.panel, -1, "Manage Files",
                               size=(285, -1))
        self.statBoxSizerMgrV = wx.StaticBoxSizer(staticBox,
                                              wx.VERTICAL)
    #-----
    def layoutWidgets(self):
        boxSizerV = wx.BoxSizer( wx.VERTICAL )
        boxSizerV.Add( self.statBoxSizerV, 1, wx.ALL )
        boxSizerV.Add( self.statBoxSizerMgrV, 1, wx.ALL )

        self.panel.SetSizer( boxSizerV )
        boxSizerV.SetSizeHints( self.panel )
    #-----
    def addFileWidgets(self):
        boxSizerH = wx.BoxSizer(wx.HORIZONTAL)
        boxSizerH.Add(wx.Button(self.panel, label='Browse to
File...'))
        boxSizerH.Add(wx.TextCtrl(self.panel, size=(174, -1),
value= "Z: "))

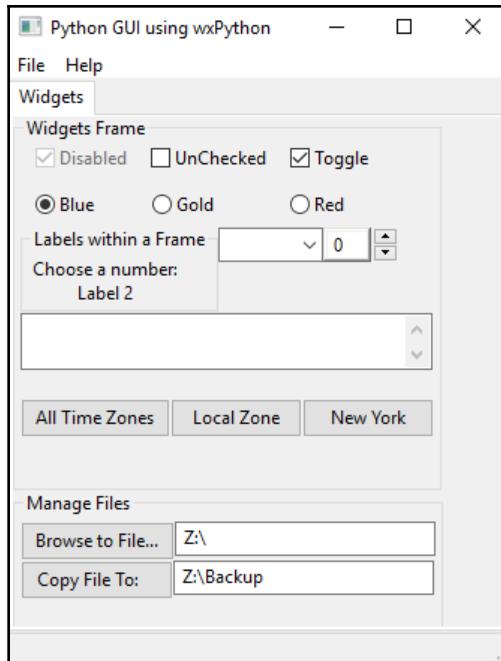
        boxSizerH1 = wx.BoxSizer(wx.HORIZONTAL)
        boxSizerH1.Add(wx.Button(self.panel, label=
'Copy File To: '))
```

```
        boxSizerH1.Add(wx.TextCtrl(self.panel, size=(174, -1),
                                value="Z:Backup" ))  
  
        boxSizerV = wx.BoxSizer(wx.VERTICAL)
        boxSizerV.Add(boxSizerH)
        boxSizerV.Add(boxSizerH1)  
  
        self.statBoxSizerMgrV.Add( boxSizerV, 1, wx.ALL )
```

9. At the bottom of the module, add the code to call MainLoop:

```
#=====
# Start GUI
#=====
app = wx.App()
MainFrame(None, , size=(350,450))
app.MainLoop()
```

10. Run `GUI_wxPython.py`. The final result of our wxPython GUI looks as follows:



Now, let's go behind the scenes to understand the code better.

How it works...

First, we create a new Python module. For reasons of clarity, we no longer call our class `OOP` but, instead, rename it as `MainFrame`.

In wxPython, the main GUI window is called a frame.



We also create a callback method that closes the GUI when we click the `Exit` menu item and declare a light-gray tuple as the background color for our GUI. We then add a tabbed control to our GUI by creating an instance of the wxPython `Notebook` class and assign it as the parent of our own custom class named `Widgets`. The `Notebook` class instance variable has `wx.Panel` as its parent.

In wxPython, the tabbed widget is named `Notebook`, just like in `tkinter`.



Every `Notebook` widget needs to have a parent and, in order to lay out widgets in the `Notebook` in wxPython, we use different kinds of sizers.

wxPython sizers are layout managers, similar to grid layout manager of `tkinter`.



Next, we add controls to our `Notebook` page, and we do this by creating a separate class, `Widgets`, that inherits from `wx.Panel`. We modularize our GUI code by breaking it into small methods, following Python OOP programming best practices, which keeps our code manageable and understandable.

When using wxPython `StaticBox` widgets, in order to successfully lay them out, we use a combination of `StaticBoxSizer` and a regular `BoxSizer`. The wxPython `StaticBox` is very similar to the `LabelFrame` widget of `tkinter`.



Embedding a `StaticBox` within another `StaticBox` is straightforward in `tkinter`, but using `wxPython` is a little non-intuitive. One way to make it work is shown in the `addStaticBoxWithLabels` method.

After this, we create a horizontal `BoxSizer`. Next, we create a vertical `StaticBoxSizer` because we want to arrange two labels in a vertical layout in this frame. In order to arrange another widget to the right of the embedded `StaticBox`, we have to assign both the embedded `StaticBox` with its children controls and the next widget to the horizontal `BoxSizer`. Next, we need to assign this `BoxSizer`, which now contains both our embedded `StaticBox` and our other widgets, to the main `StaticBox`.

Does this sound confusing?

Just experiment with these sizers to get a feel of how to use them. Start with the code for this recipe and comment out some code or modify some *x* and *y* coordinates to see the effects. It is also good to read the official `wxPython` documentation to learn more.



The important thing is knowing where to add the different sizers in the code in order to achieve the layout we wish.

In order to create the second `StaticBox` below the first, we create separate `StaticBoxSizer` and assign them to the same panel. We design and lay out our `wxPython` GUI in several classes. Once we have done this, in the bottom section of our Python module, we create an instance of the `wxPython` application. Next, we instantiate our `wxPython` GUI code.

After that, we call the main GUI event loop, which executes all of our Python code running within this application process. This displays our `wxPython` GUI.

This recipe used OOP to show how to use the `wxPython` GUI toolkit.

Trying to embed a main `wxPython` app in a main `tkinter` app

Now that we have created the same GUI using both Python's built-in `tkinter` library as well as the `wxPython` wrapper of the `wxWidgets` library, we really would like to combine the GUIs we created using these technologies.



Both the wxPython and the tkinter libraries have their own advantages.

In online forums, such as <http://stackoverflow.com/>, we often see questions such as which one is better, which GUI toolkit should I use, and so on. This suggests that we have to make an either-or decision. In reality, we do not have to make such a decision.

One of the main challenges in doing so is that each GUI toolkit must have its own event loop. In this recipe, we will try to embed a simple wxPython GUI by calling it from our tkinter GUI.

Getting ready

We will reuse the tkinter GUI we built in the *Combo box widgets* recipe in Chapter 1, *Creating the GUI Form and Adding Widgets*.

How to do it...

We will start from a simple tkinter GUI:

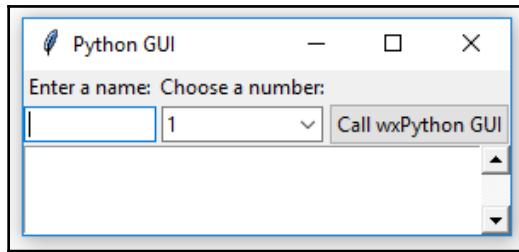
1. Create a new module and name it `Embed_wxPython.py`.
2. Add the following code:

```
#=====
import tkinter as tk
from tkinter import ttk, scrolledtext

win = tk.Tk()
win.title("Python GUI")
aLabel = ttk.Label(win, text="A Label")
aLabel.grid(column=0, row=0)
ttk.Label(win, text="Enter a name:").grid(column=0, row=0)
name = tk.StringVar()
nameEntered = ttk.Entry(win, width=12, textvariable=name)
nameEntered.grid(column=0, row=1)
ttk.Label(win, text="Choose a number:").grid(column=1, row=0)
number = tk.StringVar()
numberChosen = ttk.Combobox(win, width=12, textvariable=number)
numberChosen['values'] = (1, 2, 4, 42, 100)
numberChosen.grid(column=1, row=1)
numberChosen.current(0)
scrolW = 30
scrolH = 3
scr = scrolledtext.ScrolledText(win, width=scrolW, height=scrolH,
```

```
wrap=tk.WORD)
scr.grid(column=0, sticky='WE', columnspan=3)
nameEntered.focus()
action = ttk.Button(win, text="Call wxPython GUI")
action.grid(column=2, row=1)
#=====
# Start GUI
#=====
win.mainloop()
```

3. Run the code and observe the following output:



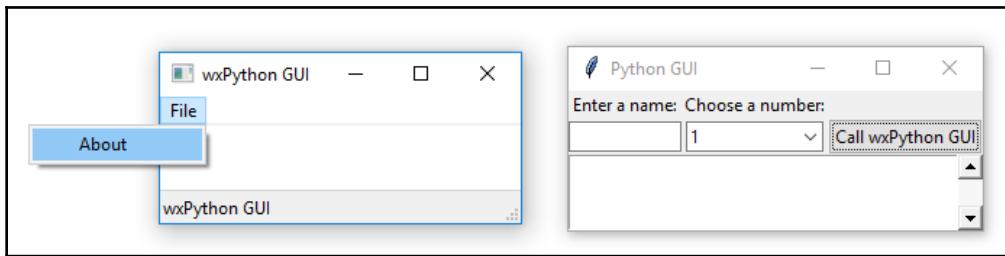
4. Create a new function, `wxPythonApp`, and place it above the main loop:

```
#=====
def wxPythonApp():
    import wx
    app = wx.App()
    frame = wx.Frame(None, -1, "wxPython GUI", size=(200,150))
    frame.SetBackgroundColour('white')
    frame.CreateStatusBar()
    menu= wx.Menu()
    menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
    menuBar = wx.MenuBar()
    menuBar.Append(menu, "File")
    frame.SetMenuBar(menuBar)
    frame.Show()
    app.MainLoop()

#===== Bottom of module =====
action = ttk.Button(win, text="Call wxPython GUI",
command=wxPythonApp)
action.grid(column=2, row=1)

#=====
# Start GUI
#=====
win.mainloop()
```

5. Running the preceding code starts a wxPython GUI from our tkinter GUI after clicking the tkinter button control:



Now, let's go behind the scenes to understand the code better.

How it works...

First, we create a simple tkinter GUI and run it by itself. Next, we try to invoke a simple wxPython GUI, which we created in a previous recipe in this chapter.

We create a new function, `wxPythonApp`, which has the wxPython code, and we place it above the tkinter button. After that, we set the `command` attribute of the button to this function:

```
action = ttk.Button(win, text="Call wxPython GUI", command=wxPythonApp) #  
<=====
```

The important part is that we place the entire wxPython code into its own function, which we named `def wxPythonApp()`. In the callback function for the button-click event, we simply call this code.

One thing to note is that we have to close the wxPython GUI before we can continue using the tkinter GUI.



Trying to embed our tkinter GUI code into wxPython

In this recipe, we will go in the opposite direction of the previous recipe and try to call our tkinter GUI code from within a wxPython GUI.

Getting ready

We will reuse some of the wxPython GUI code we created in a previous recipe in this chapter.

How to do it...

We will start from a simple wxPython GUI:

1. Create a new module and name it `Embed_tkinter.py`.
2. Add the following code:

```
#=====
import wx
app = wx.App()
frame = wx.Frame(None, -1, "wxPython GUI", size=(270,180))
frame.SetBackgroundColour('white')
frame.CreateStatusBar()
menu= wx.Menu()
menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
menuBar = wx.MenuBar()
menuBar.Append(menu,"File")
frame.SetMenuBar(menuBar)
textBox = wx.TextCtrl(frame, size=(250,50), style=wx.TE_MULTILINE)

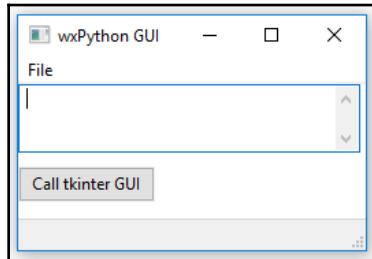
def tkinterEmbed(event):
    tkinterApp()                                     # ===== we create this
function next

button = wx.Button(frame, label="Call tkinter GUI", pos=(0,60))
frame.Bind(wx.EVT_BUTTON, tkinterEmbed, button)
frame.Show()

#=====
# Start wxPython GUI
=====
```

```
app.MainLoop()
```

3. Run the code and observe the following output:



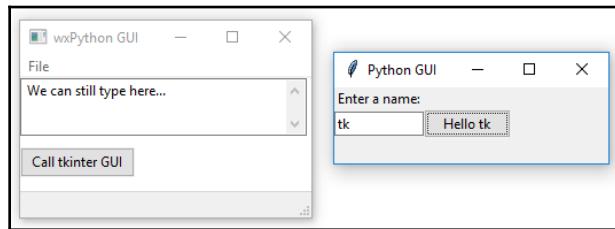
4. Add the following code to the very top of the module:

```
#=====
def tkinterApp():
    import tkinter as tk
    from tkinter import ttk
    win = tk.Tk()
    win.title("Python GUI")
    aLabel = ttk.Label(win, text="A Label")
    aLabel.grid(column=0, row=0)
    ttk.Label(win, text="Enter a name:").grid(column=0, row=0)
    name = tk.StringVar()
    nameEntered = ttk.Entry(win, width=12, textvariable=name)
    nameEntered.grid(column=0, row=1)
    nameEntered.focus()

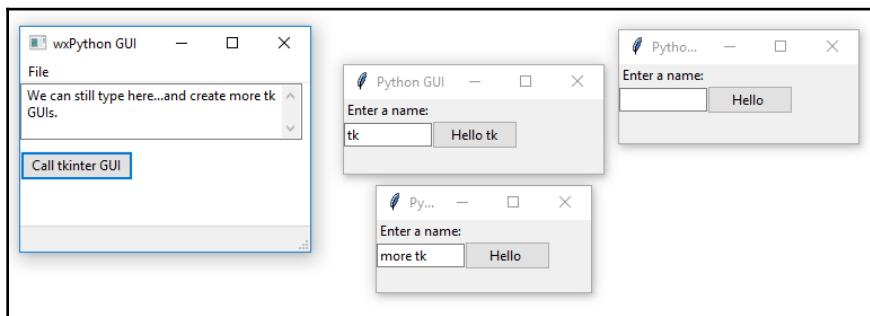
    def buttonCallback():
        action.configure(text='Hello ' + name.get())
    action = ttk.Button(win, text="Print", command=buttonCallback)
    action.grid(column=2, row=1)
    win.mainloop()
#===== Bottom of module =====
import wx
# ...
```

5. Run the code and click the **Call tkinter GUI** button.
6. In the **tkinter GUI**, enter some text and click the **Print** button.

7. Type into the wxPython `TextCtrl` widget:



8. Run the code and click the **Call tkinter GUI** button several times.
9. Type into the tkinter GUIs and click the **Print** button:



Now, let's go behind the scenes to understand the code better.

How it works...

In this recipe, we went in the opposite direction of the previous recipe by first creating a GUI using wxPython and then, from within it, creating several GUI instances built using tkinter.

Running the `Embed_tkinter.py` code starts a tkinter GUI from our wxPython GUI after clicking the wxPython button widget. We can then enter text into the tkinter textbox and, by clicking its button, the button text gets updated with the name.

After starting the tkinter event loop, the wxPython GUI is still responsive because we can still type into the `TextCtrl` widget while the tkinter GUI is up and running.



In the previous recipe, we could not use our `tkinter` GUI until we had closed the `wxPython` GUI. Being aware of this difference can help our design decisions if we want to combine the two Python GUI technologies.

We can also create several `tkinter` GUI instances by clicking the `wxPython` GUI several times. We cannot, however, close the `wxPython` GUI while any `tkinter` GUIs are still running. We have to close them first.

The `wxPython` GUI remained responsive while one or more `tkinter` GUIs were running. However, clicking the `tkinter` button only updated its button text in the first instance.

Using Python to control two different GUI frameworks

In this recipe, we will explore ways to control the `tkinter` and `wxPython` GUI frameworks from Python. We have already used the Python threading module to keep our GUI responsive in [Chapter 6, Threads and Networking](#), so here we will attempt to use the same approach.

We will see that things don't always work in a way that would be intuitive. However, we will improve our `tkinter` GUI from being unresponsive while we invoke an instance of the `wxPython` GUI from within it.

Getting ready

This recipe will extend a previous recipe from this chapter, *Trying to embed a main wxPython app in a main tkinter app*, in which we successfully tried to embed a main `wxPython` GUI into our `tkinter` GUI.

How to do it...

When we created an instance of a `wxPython` GUI from our `tkinter` GUI, we could no longer use the `tkinter` GUI controls until we closed the one instance of the `wxPython` GUI. Let's improve on this now:

1. Create a new module and name it `Control_Frameworks_NOT_working.py`.

2. Write the following code:

```
#=====
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from threading import Thread

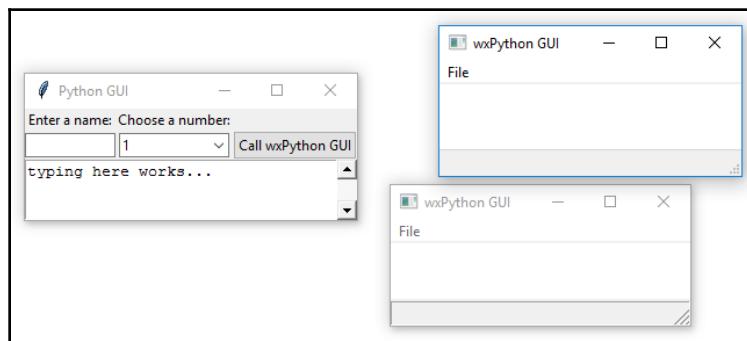
win = tk.Tk()
win.title("Python GUI")
aLabel = ttk.Label(win, text="A Label")
aLabel.grid(column=0, row=0)
ttk.Label(win, text="Enter a name:").grid(column=0, row=0)
name = tk.StringVar()
nameEntered = ttk.Entry(win, width=12, textvariable=name)
nameEntered.grid(column=0, row=1)
ttk.Label(win, text="Choose a number:").grid(column=1, row=0)
number = tk.StringVar()
numberChosen = ttk.Combobox(win, width=12, textvariable=number)
numberChosen['values'] = (1, 2, 4, 42, 100)
numberChosen.grid(column=1, row=1)
numberChosen.current(0)
scrolW = 30
scrolH = 3
scr = scrolledtext.ScrolledText(win, width=scrolW, height=scrolH,
wrap=tk.WORD)
scr.grid(column=0, sticky='WE', columnspan=3)
nameEntered.focus()

#=====
# NOT working - CRASHES Python -----
def wxPythonApp():
    import wx
    app = wx.App()
    frame = wx.Frame(None, -1, "wxPython GUI", size=(200,150))
    frame.SetBackgroundColour('white')
    frame.CreateStatusBar()
    menu= wx.Menu()
    menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
    menuBar = wx.MenuBar()
    menuBar.Append(menu, "File")
    frame.SetMenuBar(menuBar)
    frame.Show()
    app.MainLoop()

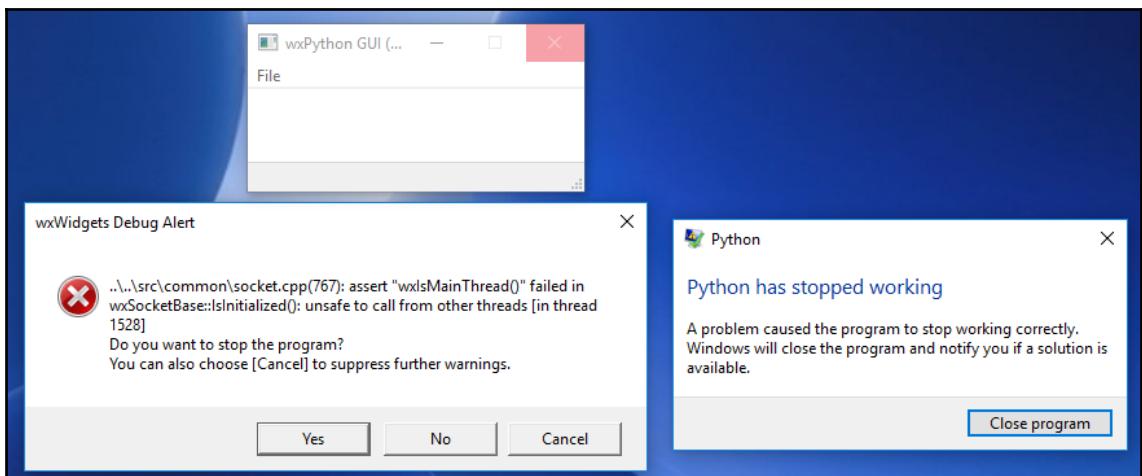
def tryRunInThread():
    runT = Thread(target=wxPythonApp)           # ===== calling
wxPythonApp in thread
```

```
runT.setDaemon(True)
runT.start()
print(runT)
print('createThread():', runT.isAlive())
action = ttk.Button(win, text="Call wxPython GUI",
command=tryRunInThread)
action.grid(column=2, row=1)
#-----
#=====
# Start GUI
#=====
win.mainloop()
```

3. Run the code. Open some wxPython GUIs and type into the tkinter GUI:



4. Close the GUIs:



In order to avoid this crashing of the Python.exe executable process as shown in the preceding screenshot, instead of trying to run the entire wxPython application in a thread, we can change the code to make only the wxPython app.MainLoop run in a thread.

5. Create a new module and name it Control_Frameworks.py.
6. Write the following code and run it:

```
#=====
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from threading import Thread

win = tk.Tk()
win.title("Python GUI")
aLabel = ttk.Label(win, text="A Label")
aLabel.grid(column=0, row=0)
ttk.Label(win, text="Enter a name:").grid(column=0, row=0)
name = tk.StringVar()
nameEntered = ttk.Entry(win, width=12, textvariable=name)
nameEntered.grid(column=0, row=1)
ttk.Label(win, text="Choose a number:").grid(column=1, row=0)
number = tk.StringVar()
numberChosen = ttk.Combobox(win, width=12, textvariable=number)
numberChosen['values'] = (1, 2, 4, 42, 100)
numberChosen.grid(column=1, row=1)
numberChosen.current(0)
scrolW = 30
scrolH = 3
scr = scrolledtext.ScrolledText(win, width=scrolW, height=scrolH,
wrap=tk.WORD)
scr.grid(column=0, sticky='WE', columnspan=3)
nameEntered.focus()

#=====
## working
def wxPythonApp():
    import wx
    app = wx.App()
    frame = wx.Frame(None, -1, "wxPython GUI", size=(200,150))
    frame.SetBackgroundColour('white')
    frame.CreateStatusBar()
    menu= wx.Menu()
    menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
    menuBar = wx.MenuBar()
    menuBar.Append(menu, "File")
```

```
frame.SetMenuBar(menuBar)
frame.Show()
runT = Thread(target=app.MainLoop)           # <==== Thread for
MainLoop only
runT.setDaemon(True)
runT.start()
print(runT)
print('createThread():', runT.isAlive())

action = ttk.Button(win, text="Call wxPython GUI",
command=wxPythonApp)
action.grid(column=2, row=1)
=====
# Start GUI
=====
win.mainloop()
```

Let's comprehend the steps in detail in the next section.

How it works...

We first tried to run the entire wxPython GUI application in a thread, `Control_Frameworks_NOT_working.py`, but this did not work because the wxPython main event loop expects to be the main thread of the application.

At first, `Control_Frameworks_NOT_working.py` seems to be working, which would be intuitive, as the tkinter controls are no longer disabled and we can create several instances of the wxPython GUI by clicking the button. We can also type into the wxPython GUI and select the other tkinter widgets. However, once we try to close the GUIs, we get an error from wxWidgets and our Python executable crashes.

We found a workaround, `Control_Frameworks.py`, for this by only running the wxPython `app.MainLoop` in a thread that tricks it into believing it is the main thread. One side effect of this approach is that we can no longer individually close all of the wxPython GUI instances. At least one of them only closes when we close the wxPython GUI that created the threads as daemons. You can test this out by clicking the **Call wxPython GUI** button once or several times and then try to close all the created wxPython window forms. We cannot close the last one until we close the calling tkinter GUI!

I am not quite sure why this is. Intuitively, we might expect to be able to close all daemon threads without having to wait for the main thread that created them to close first. It possibly has to do with a reference counter not having been set to zero while our main thread is still running. On a pragmatic level, this is how it currently works.

Communicating between the two connected GUIs

In this recipe, we will explore ways to make the two GUIs talk to each other.

Getting ready

Reading the previous recipe might be a good preparation for this one.

In this recipe, we will use the slightly modified GUI code with respect to the previous recipe, but most of the basic GUI-building code is the same.

How to do it...

We will write Python code that makes the two GUIs communicate with each other to a certain degree:

1. Create a new module and name it `Communicate.py`.
2. Add the following code:

```
#=====
import tkinter as tk
from tkinter import ttk
from threading import Thread

win = tk.Tk()
win.title("Python GUI")
ttk.Label(win, text="Enter a name:").grid(column=0, row=0)

name = tk.StringVar()
nameEntered = ttk.Entry(win, width=12, textvariable=name)
nameEntered.grid(column=0, row=1)
nameEntered.focus()

ttk.Label(win, text="Choose a number:").grid(column=1, row=0)
```

```
number = tk.StringVar()
numberChosen = ttk.Combobox(win, width=12, textvariable=number)
numberChosen['values'] = (1, 2, 4, 42, 100)
numberChosen.grid(column=1, row=1)
numberChosen.current(0)

text = tk.Text(win, height=10, width=40, borderwidth=2,
wrap='word')
text.grid(column=0, sticky='WE', columnspan=3)

#=====
from multiprocessing import Queue
sharedQueue = Queue()
dataInQueue = False

def putDataIntoQueue(data):
    global dataInQueue
    dataInQueue = True
    sharedQueue.put(data)
def readDataFromQueue():
    global dataInQueue
    dataInQueue = False
    return sharedQueue.get()
#=====

import wx
class GUI(wx.Panel):
    def __init__(self, parent):
        wx.Panel.__init__(self, parent)
        parent.CreateStatusBar()
        menu= wx.Menu()
        menu.Append(wx.ID_ABOUT, "About", "wxPython GUI")
        menuBar = wx.MenuBar()
        menuBar.Append(menu, "File")
        parent.SetMenuBar(menuBar)
        button = wx.Button(self, label="Print", pos=(0,60))
        self.Bind(wx.EVT_BUTTON, self.writeToSharedQueue, button)
        self.textBox = wx.TextCtrl(self, size=(280,50),
style=wx.TE_MULTILINE)

    #-----
    ---#
    def writeToSharedQueue(self, event):
        self.textBox.AppendText(
            "The Print Button has been clicked!\n")
        putDataIntoQueue('Hi from wxPython via Shared Queue.\n')
        if dataInQueue:
            data = readDataFromQueue()
            self.textBox.AppendText(data)
```

```
text.insert('0.0', data) # insert data into tkinter GUI

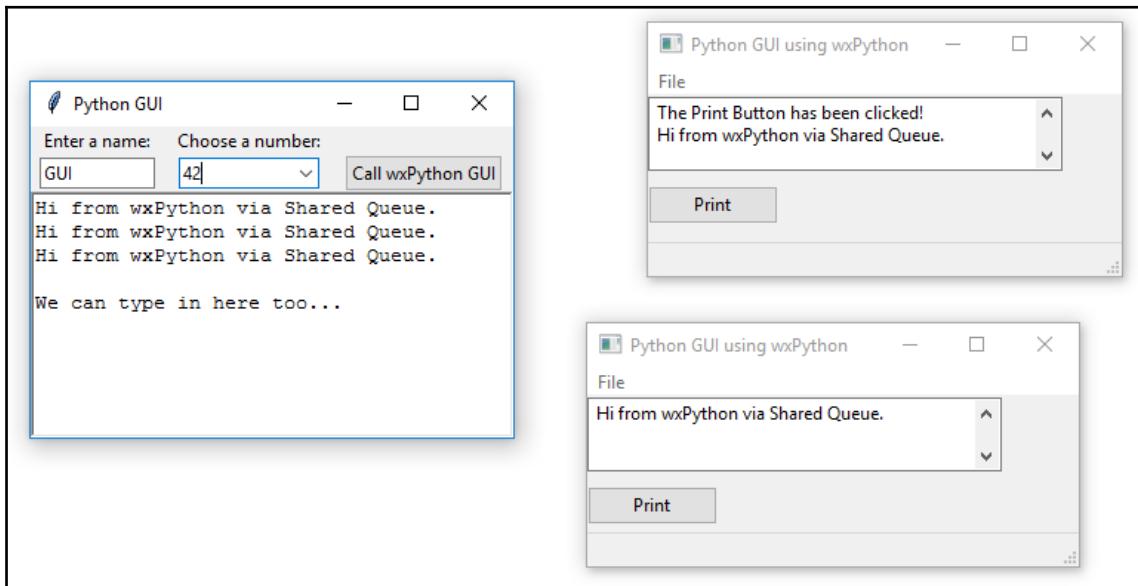
=====
def wxPythonApp():
    app = wx.App()
    frame = wx.Frame(
        None, title="Python GUI using wxPython",
        size=(300,180))
    GUI(frame)
    frame.Show()
    runT = Thread(target=app.MainLoop)
    runT.setDaemon(True)
    runT.start()
    print(runt)
    print('createThread():', runT.isAlive())
=====

action = ttk.Button(win, text="Call wxPython GUI",
command=wxPythonApp)
action.grid(column=2, row=1)

=====
# Start GUI
=====
win.mainloop()
```

3. Run the code and then perform the next step.

4. Click both buttons and type into the controls:



Now, let's go behind the scenes to understand the code better.

How it works...

Running the `Communicate.py` code first creates the `tkinter` part of the program, and when we click the button in this GUI, it runs the `wxPython` GUI. Both are running at the same time as before, but this time there is an extra level of communication between the two GUIs.

The `tkinter` GUI is shown on the left-hand side in the preceding screenshot, and by clicking the **Call wxPython GUI** button, we invoke an instance of the `wxPython` GUI. We can create several instances by clicking the button several times.

All of the created GUIs remain responsive. They do not crash nor freeze.



Clicking the **Print** button on any of the wxPython GUI instances writes one sentence to its own `TextCtrl` widget and then writes another line to itself, as well as to the `tkinter` GUI. You will have to scroll up to see the first sentence in the wxPython GUI.



The way this works is by using a module-level queue and a `Text` widget of `tkinter`.

One important element to note is that we create a thread to run the wxPython `app.MainLoop`, as we did in the previous recipe:

```
def wxPythonApp():
    app = wx.App()
    frame = wx.Frame(None, title="Python GUI using wxPython",
                     size=(300,180))
    GUI(frame)
    frame.Show()
    runT = Thread(target=app.MainLoop)
    runT.setDaemon(True)
    runT.start()
```

We created a class that inherits from `wx.Panel` and named it `GUI` and then instantiated an instance of this class in the preceding code.

We created a button-click event callback method in this class, which then calls the code that was written above it. Because of this, the class has access to the functions and can write to the shared queue:

```
#-----
def writeToSharedQueue(self, event):
    self.textBox.AppendText("The Print Button has been clicked!\n")
    putDataIntoQueue('Hi from wxPython via Shared Queue.\n')
    if dataInQueue:
        data = readDataFromQueue()
        self.textBox.AppendText(data)
        text.insert('0.0', data) # insert data into tkinter
```

We first check whether the data has been placed in the shared queue in the preceding method and, if that is the case, print the common data to both GUIs.



The `putDataIntoQueue()` line places data into the queue and `readDataFromQueue()` reads it back out, saving it in the `data` variable. `text.insert('0.0', data)` is the line that writes this data into the `tkinter` GUI from the **Print** button's `wxPython` callback method.

The following functions are called in the code and make it work:

```
from multiprocessing import Queue
sharedQueue = Queue()
dataInQueue = False

def putDataIntoQueue(data):
    global dataInQueue
    dataInQueue = True
    sharedQueue.put(data)

def readDataFromQueue():
    global dataInQueue
    dataInQueue = False
    return sharedQueue.get()
```

We used a simple boolean flag named `dataInQueue` to communicate when the data is available in the queue.

In this recipe, we have successfully combined the two GUIs we created in a similar fashion, but which were previously standalone and not talking to each other. However, in this recipe, we connected them further by making one GUI launch another and, via a simple `multiprocessing` Python queue mechanism, we were able to make them communicate with each other, writing data from a shared queue into both GUIs.



There are many more advanced and complicated technologies available to connect different processes, threads, pools, locks, pipes, TCP/IP connections, and so on.

In the Pythonic spirit, we found a simple solution that works for us. Once our code becomes more complicated, we might have to refactor it, but this is a good beginning.

10

Building GUIs with PyQt5

In this chapter, we will introduce another Python GUI toolkit, named PyQt5, which is truly excellent. PyQt5 has similar capabilities to `tkinter` but comes with a very nice Visual Designer tool that lets us drag and drop widgets onto a form. We will also use another tool that converts the Designer `.ui` code into Python code.

After visually designing our GUI in the Designer and then converting the code into Python code, we will continue using pure Python to add functionality to our widgets. First, we will install PyQt5 and the Designer before writing a simple PyQt5 GUI without the Designer. After that, we will visually design our GUI.

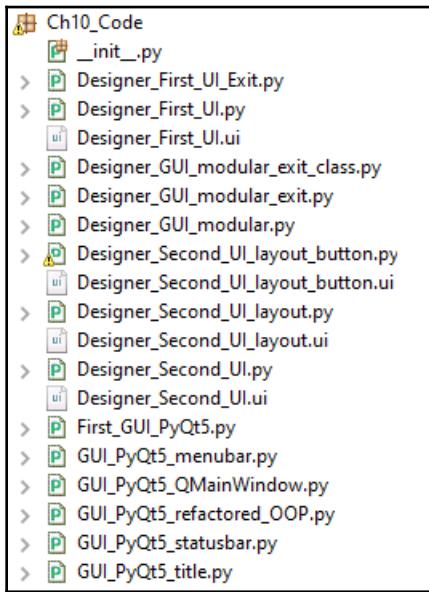
Knowing how to use PyQt5, and the Visual Designer tool and how to convert `.ui` into `.py` code will add great skills to your Python GUI development toolbox. From this, you will learn how to create powerful and complex GUIs, as well as how to visually design the UI and then decouple the functionality from the design using a modular approach to software development.

This also gives you the opportunity to compare the different GUI frameworks we have been showing you throughout this book, which will ultimately lead to you choosing one to explore in more depth.



I have created two Packt video courses that focus very deeply on Python GUI programming with `tkinter` and **PyQt5**. You can find them on the Packt website. I will also provide links to them at the end of this chapter.

The following screenshot provides an overview of the Python modules you will need for this chapter:



We will be covering the following recipes:

- Installing PyQt5
- Installing the PyQt5 Designer tool
- Writing our first PyQt5 GUI
- Changing the title of the GUI
- Refactoring our code with object-oriented programming
- Inheriting from QMainWindow
- Adding a status bar widget
- Adding a menu bar widget
- Starting the PyQt5 Designer tool
- Previewing the form within the PyQt5 Designer

- Saving the PyQt5 Designer form
- Converting Designer .ui code into .py code
- Understanding the converted Designer code
- Building a modular GUI design
- Adding another menu item to our menu bar
- Connecting functionality to the Exit menu item
- Adding a Tab Widget via the Designer
- Using layouts in the Designer
- Adding buttons and labels in the Designer

Installing PyQt5

In this recipe, we will install the PyQt5 GUI framework. We will be using Python's `pip` tool to download the PyQt5 wheel format installer.

You can find the official documentation at the following link: <https://www.riverbankcomputing.com/static/Docs/PyQt5/installation.html>.

Getting ready

You need to have Python's `pip` tool installed on your computer. You probably already have it.

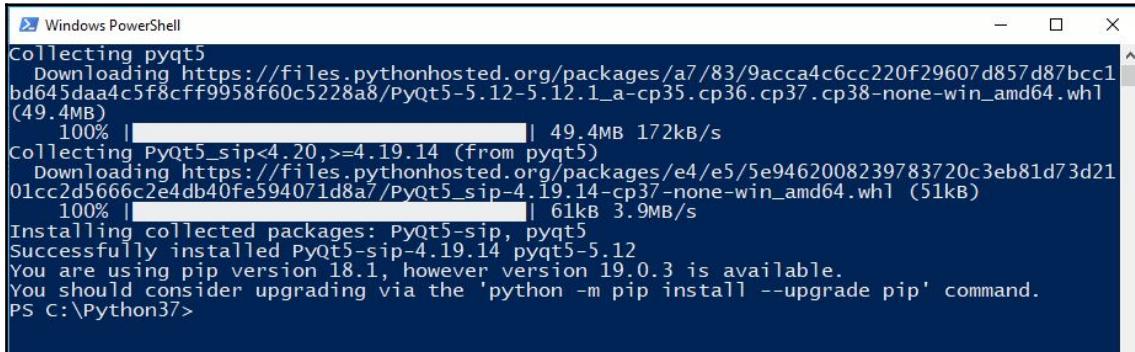
How to do it...

Let's see how we can install PyQt5 using Python's `pip` tool:

1. Open a Windows PowerShell window or Command Prompt.
2. Type in the `pip install pyqt5` command.
3. Press the *Enter* key.
4. Verify the installation by running `pip list`.

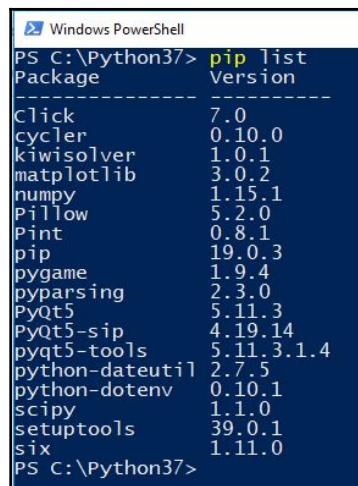
How it works...

In *step 1*, we open a PowerShell window, and in *step 2*, we use Python's `pip` tool. After pressing the *Enter* key to run the command in *step 3*, the installation will start and run to completion. You will see output similar to the following:



```
Windows PowerShell
Collecting pyqt5
  Downloading https://files.pythonhosted.org/packages/a7/83/9acca4c6cc220f29607d857d87bcc1bd645daa4c5f8cff9958f60c5228a8/PyQt5-5.12.1_a-cp35.cp36.cp37.cp38-none-win_amd64.whl (49.4MB)
    100% |██████████| 49.4MB 172kB/s
Collecting PyQt5_sip<4.20,>=4.19.14 (from pyqt5)
  Downloading https://files.pythonhosted.org/packages/e4/e5/5e9462008239783720c3eb81d73d2101cc2d5666c2e4db40fe594071d8a7/PyQt5_sip-4.19.14-cp37-none-win_amd64.whl (51kB)
    100% |██████████| 61kB 3.9MB/s
Installing collected packages: PyQt5-sip, pyqt5
Successfully installed PyQt5-sip-4.19.14 pyqt5-5.12
You are using pip version 18.1, however version 19.0.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
PS C:\Python37>
```

In *step 4*, we use `pip` again to verify that we have installed PyQt5 successfully. The output will look similar to the following screenshot:



```
Windows PowerShell
PS C:\Python37> pip list
Package           Version
-----
click             7.0
cycler            0.10.0
kiwisolver        1.0.1
matplotlib        3.0.2
numpy              1.15.1
pillow             5.2.0
pint               0.8.1
pip                19.0.3
pygame             1.9.4
pyparsing          2.3.0
PyQt5              5.11.3
PyQt5-sip         4.19.14
PyQt5-tools       5.11.3.1.4
python-dateutil    2.7.5
python-dotenv      0.10.1
scipy              1.1.0
setuptools         39.0.1
six                1.11.0
PS C:\Python37>
```

You may see more packages installed on your computer. The important thing to check is that the PyQt5 package is listed. The version number that's been installed is listed to the right of the package's name.

Installing the PyQt5 Designer tool

In this recipe, we will install the PyQt5 Designer tool. We will do this by using Python's `pip` tool. The steps are very similar to the previous recipe's, where we installed PyQt5.

Getting ready

You need to have Python's `pip` tool installed on your computer.

How to do it...

Let's see how we can install the PyQt5 Designer using Python's `pip` tool. Note that the package includes more than just the Designer tool:

1. Open a Windows PowerShell window or Command Prompt.
2. Type in the following command:

```
pip install pyqt5-tools
```

3. Press the *Enter* key.
4. Verify the installation by running the following command:

```
pip list
```

5. Locate the `Designer.exe` file on your hard drive.

How it works...

In *step 1*, we open a PowerShell window and in *step 2*, we use Python's pip tool. After pressing the *Enter* key to run the command in *step 3*, the installation will start and run to completion. You will see output similar to the following:

```
PS C:\Python37> pip install pyqt5-tools
Collecting pyqt5-tools
  Using cached https://files.pythonhosted.org/packages/6a/35/98279ba706e9c3731f0f53813e9a8152ed662f2230e8af18a20d490167be/pyqt5==5.11.3.1.4-cp37-none-win_amd64.whl (59.4MB)
    100% |████████████████████████████████| 59.4MB 13kB/s
Collecting pyqt5==5.11.3 (from pyqt5-tools)
  Downloading https://files.pythonhosted.org/packages/a7/2d/d2c989006c86ae98ed230c28c3e0dd7fa0374e723afc107d12268159ceb7/PyQt5-5.11.3-5.11.2-cp35.cp36.cp37.cp38-none-win_amd64.whl (93.4MB)
    100% |████████████████████████████████| 93.4MB 78kB/s
Collecting click (from pyqt5-tools)
  Downloading https://files.pythonhosted.org/packages/fa/37/45185cb5abbc30d7257104c434fe0b07e5a195a6847506c074527aa599ec/Click-7.0-py2.py3-none-any.whl (81kB)
    100% |████████████████████████████████| 81kB 5.3MB/s
Collecting python-dotenv (from pyqt5-tools)
  Downloading https://files.pythonhosted.org/packages/8c/14/501508b016e7b1ad0eb91bba581e66ad9bfc7c66fcacbb580eaf9bc38458/python_dotenv-0.10.1-py2.py3-none-any.whl
Requirement already satisfied: PyQt5_sip<4.20,>=4.19.11 in c:\python37\lib\site-packages (from pyqt5==5.11.3->pyqt5-tools) (4.19.14)
Installing collected packages: pyqt5, click, python-dotenv, pyqt5-tools
  Found existing installation: PyQt5 5.12
    Uninstalling PyQt5-5.12:
      Successfully uninstalled PyQt5-5.12

PS C:\Python37>
```

```
PS C:\Python37> pip install pyqt5-tools
Collecting pyqt5-tools
  Using cached https://files.pythonhosted.org/packages/6a/35/98279ba706e9c3731f0f53813e9a8152ed662f2230e8af18a20d490167be/pyqt5==5.11.3.1.4-cp37-none-win_amd64.whl
Requirement already satisfied: pyqt5==5.11.3 in c:\python37\lib\site-packages (from pyqt5-tools) (5.11.3)
Collecting click (from pyqt5-tools)
  Using cached https://files.pythonhosted.org/packages/fa/37/45185cb5abbc30d7257104c434fe0b07e5a195a6847506c074527aa599ec/Click-7.0-py2.py3-none-any.whl
Collecting python-dotenv (from pyqt5-tools)
  Using cached https://files.pythonhosted.org/packages/8c/14/501508b016e7b1ad0eb91bba581e66ad9bfc7c66fcacbb580eaf9bc38458/python_dotenv-0.10.1-py2.py3-none-any.whl
Requirement already satisfied: PyQt5_sip<4.20,>=4.19.11 in c:\python37\lib\site-packages (from pyqt5==5.11.3->pyqt5-tools) (4.19.14)
Installing collected packages: click, python-dotenv, pyqt5-tools
Successfully installed click-7.0 pyqt5-tools-5.11.3.1.4 python-dotenv-0.10.1
PS C:\Python37>
```

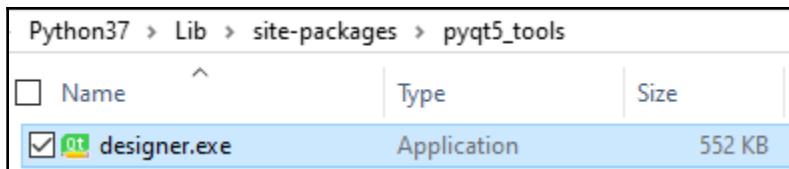


Please note that, in the preceding screenshot, the installation ran into an error. I don't know why, but sometimes installations do run into errors. I simply reran the installation, and this time it ran without any errors. The necessary tools, including the Designer, were installed successfully.

Step 4 is the exact same step as in the previous recipe, and the output is exactly the same. Please refer to the output screenshot of the *Installing PyQt5* recipe for more information.

In *step 5*, we want to find the `Designer.exe` file, which is the Visual Designer tool we will use in later recipes. After finding it, you will want to make a shortcut to it on your desktop.

Here is a screenshot of where `Designer.exe` is installed on my computer:



Your location might be different, but this gives you an idea of where to look for the tool.

Writing our first PyQt5 GUI

In this recipe, we will be writing our first PyQt5 GUI. We will be using PyQt5 directly without using the Designer.

Getting ready

You need to have PyQt5 installed. See the *Installing PyQt5* recipe to find out how to install PyQt5. Use your favorite Python editor to write the code. If you are not familiar with modern IDEs such as Eclipse, PyCharm, and so on, you can use the IDLE editor, which ships with Python.

How to do it...

Let's look at how we can build our first GUI with PyQt5:

1. Open your favorite Python editor.
2. Create a new Python module and save it as `First_GUI_PyQt5.py`.
3. Start by typing in the following import statements:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
```

4. Add the following four lines of code below the import statements:

```
app = QApplication(sys.argv)
gui = QWidget()
gui.show()
sys.exit(app.exec_())
```

5. Run the preceding code. Maximize, minimize, and resize the resultant GUI. Click the X symbol in the top-right corner to close the application:



Now, let's go behind the scenes to understand how this works.

How it works...

In *steps 1* and *2*, we create a new Python module. In *step 3*, we write some import statements.

We import `sys` so that we can pass command-line arguments into our GUI.

From the PyQt5 package, we import the `QApplication` and `QWidget` classes, both of which reside within the `QtWidgets` module.

We create an instance of the `QApplication` class, passing in `sys.argv` so that we can pass in additional command-line arguments. We save this instance in the `app` variable. This will create our application.

Then, we create an instance of the `QWidget` class, which becomes our GUI. We save this instance in a local variable named `gui`.

Next, we call the `show` method on our `gui` class instance to make the GUI visible.

After that, we call the `exec_` method on our application class instance, which executes our application. We wrap the call into `sys.exit` in order to catch any exceptions that might occur. If an exception occurs, this will make sure that our Python application exits cleanly and does not crash.

Changing the title of the GUI

In this recipe, we will change the title of the GUI we created in the previous recipe.

Getting ready

We will be using the code from the previous recipe, so either type it into a module of your own or download it from the Packt website for this book.

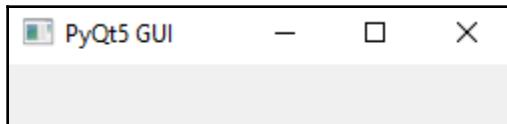
How to do it...

We will enhance the GUI from the previous recipe by changing the title of this GUI. Let's get started:

1. Open `First_GUI_PyQt5.py` and save it as `GUI_PyQt5_title.py`.
2. Add the following line of code into the middle of the existing code:

```
gui.setWindowTitle('PyQt5 GUI')
```

3. Run the code and note the new title:



Now, let's go behind the scenes to understand how this works.

How it works...

In *step 1*, we are reusing the code from the previous recipe by saving it under a new name.

In *step 2*, we are calling the `setWindowTitle` method on our `gui` instance, passing it as a string. This string becomes our new title when we run the application.

The complete code now looks like this:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv)
gui = QWidget()
gui.setWindowTitle('PyQt5 GUI')           # <-- method call in the middle
gui.show()
sys.exit(app.exec_())
```

In *step 3*, we run the code and see that our window title now displays **PyQt5 GUI** instead of **python**.

There's more...

One very important thing to note in the preceding code is the place where we are calling `setWindowTitle`, because this shows us the typical code structure every PyQt5 application follows.

After the import statements, at the top, we create a PyQt5 application. At the bottom, we execute the application. All of the functionality we add to the GUI resides in between the top and bottom pieces of code.

Refactoring our code into object-oriented programming

In this recipe, we will refactor our code into **object-oriented programming (OOP)** using classes. This is in preparation for the PyQt5 Designer code and the recipes we will be building later in this chapter. In this recipe, the resultant output of the GUI will look the same, but the code will be different.

We will build a class that inherits from `QWidget`.

Getting ready

We will be refactoring the code from the previous recipe, so make sure you understand that code.

How to do it...

We will turn our previous, procedural code into object-oriented code. Here is how we do this:

1. Create a new module and name it `GUI_PyQt5_refactored_OOP.py`.
2. Start by writing the same import statements:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
```

3. Create a class that inherits from `QWidget`:

```
class GUI(QWidget):
    def __init__(self):
        super().__init__()
        # initialize super class, which creates the Window
        self.initUI()

    def initUI(self):
        self.setWindowTitle('PyQt5 GUI')
```

4. Add a Python self-testing section under the preceding code:

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    gui = GUI()
    gui.show()
    sys.exit(app.exec_())
```

5. Run the application. The resultant GUI will be identical to the one from the previous recipe.

How it works...

In *step 1*, we are creating a new module, while in *step 2*, we are adding the same import statements we used in the previous recipes in this chapter.

In *step 3*, we are creating a new class that inherits from `QWidget`. We call `super` to initialize the parent, which, in turn, creates our GUI.

Then, we create and call a class method that sets the window title.

In *step 4*, we are using Python's self-testing capabilities to create the PyQt5 application and the GUI, and then we are executing the code.

Running this code creates the same GUI as in the previous recipe, but our code is now using OOP.

Inheriting from QMainWindow

Now that we have seen how to inherit from PyQt5 classes, in this recipe, we will inherit from `QMainWindow`. This gives us more options when it comes to designing our GUI compared to inheriting from `Widgets`. In addition to setting the GUI window title, we will also give it a certain size.

Getting ready

Read through the previous recipe so that you understand the code we are writing here.

How to do it...

We will inherit from `QMainWindow` and specify the size of the GUI. Let's get started:

1. Create a new module and name it `GUI_PyQt5_QMainWindow.py`.
2. Write the following import statements:

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow
```

3. Create the following class:

```
class GUI(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.setWindowTitle('PyQt5 GUI')
        self.resize(400, 300)
```

4. Add the same '`__main__`' code that was shown in the previous recipe.
5. Run the code. The resultant GUI will look the same as it did in the previous two recipes, but it will be smaller.

How it works...

In *step 1*, we are creating a new module, while in *step 2*, we are writing the import statements. This time, however, we are not importing `QWidgets` – we are importing `QMainWindow` instead. In *step 3*, we are creating a new class that inherits from `QMainWindow`. As before, we set the title in the method we are calling. However, in addition to setting the title, we are also giving our GUI a specific size. We do this by calling the `resize` method, passing in the width and height.

Steps 4 and *5* are the same as they were in the previous recipe, but the resultant GUI is now the size we specified in the `resize` method.

Adding a status bar widget

In this recipe, we will start to add widgets to the GUI we created previously. We will start by adding a status bar. This is a widget that comes built in with PyQt5, so all we have to do is use it.

Getting ready

We will extend the GUI from the previous recipe, so read the previous recipe in order to understand the code we are writing here.

How to do it...

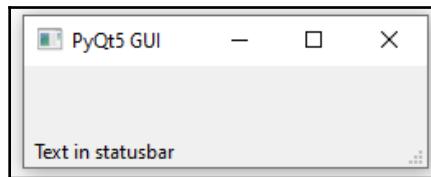
Let's get started:

1. Create a new module and name it `GUI_PyQt5_statusbar.py`.
2. Write the exact same code from the previous recipe, which can be found in `GUI_PyQt5_QMainWindow.py`.

3. Create a new method, `add_widgets`, within the class and call it, as shown in the following code block:

```
Def initUI(self):  
    self.setWindowTitle('PyQt5 GUI')  
    self.resize(400, 300)  
    self.add_widgets() # <== call new method here  
  
def add_widgets(self):  
    self.statusBar().showMessage('Text in statusbar')
```

4. Run the preceding code and note the new status bar at the bottom of the GUI:



Now, let's go behind the scenes to understand how this works.

How it works...

In *step 1*, we are creating a new module, while in *step 2*, we are reusing the code from the previous recipe. In *step 3*, we create a new method, `add_widget`, in which we are creating the PyQt5 built-in status bar. We are using `self` to access this widget since the `statusBar` widget is part of `QMainWindow`. This is one of the reasons we are inheriting from `QMainWindow` instead of `QWidget`s to build our GUI.

After creating the status bar, we immediately call the `showMessage` method on it. We could have done this in two steps, that is, creating the status bar and saving the instance of this class in a local variable, and then using the variable to call `showMessage` on it. Here, we streamlined the code into one line.

Adding a menu bar widget

In this recipe, we will add a menu bar to the GUI we created in the previous recipe. We did this in a previous chapter with `tkinter`, but in this recipe, we will see how creating a menu bar with PyQt5 is much simpler and more intuitive.

We will also start creating PyQt5 **actions**, which add functionality to the GUI.

Getting ready

We will extend the GUI from the previous recipe, where we added a status bar. Read the previous recipe in order to understand the code we are writing here.

How to do it...

We will extend from the previous recipe, in which we added our first widget. Let's see how we can do that:

1. Create a new module and name it `GUI_PyQt5_menubar.py`.
2. Copy the code from the previous recipe, which can be found in `GUI_PyQt5_statusbar.py`.
3. In the `add_widgets` method, add the following code:

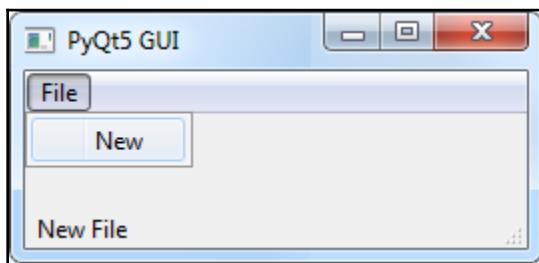
```
def add_widgets(self):
    self.statusBar().showMessage('Text in statusbar')

    menubar = self.menuBar()
    file_menu = menubar.addMenu('File')

    new_action = QAction('New', self)
    file_menu.addAction(new_action)

    new_action.setStatusTip('New File')
```

4. Run the preceding code. You will see a new menu bar with a menu item. Click on the **File** menu and then click on **New**. Look at the text in the status bar:



Let's go behind the scenes to understand how this works.

How it works...

In *step 1*, we are creating a new module, while in *step 2*, we are reusing the code from the previous recipe. In *step 3*, we add new code to the `add_widgets` method. Again, we are using `self` to access the `menuBar` class that is built into `QMainWindow`. After creating an instance of the menu bar, we use the `addMenu` method to create a menu. We use the `QAction` class to create a menu item and then we use the `addAction` method to add this menu item to the menu.

We use the `new_action` variable to call `setStatusTip`. Now, when we click on **File** | **New**, we can see the text displayed in the status bar, as shown in *step 4*.

Starting the PyQt5 Designer tool

In this recipe, we will start to use the PyQt5 Designer tool. We will visually design our GUIs and drag and drop our widgets onto a window main form. This form can be a `QWidget`s form or a `QMainWindow` form.

Getting ready

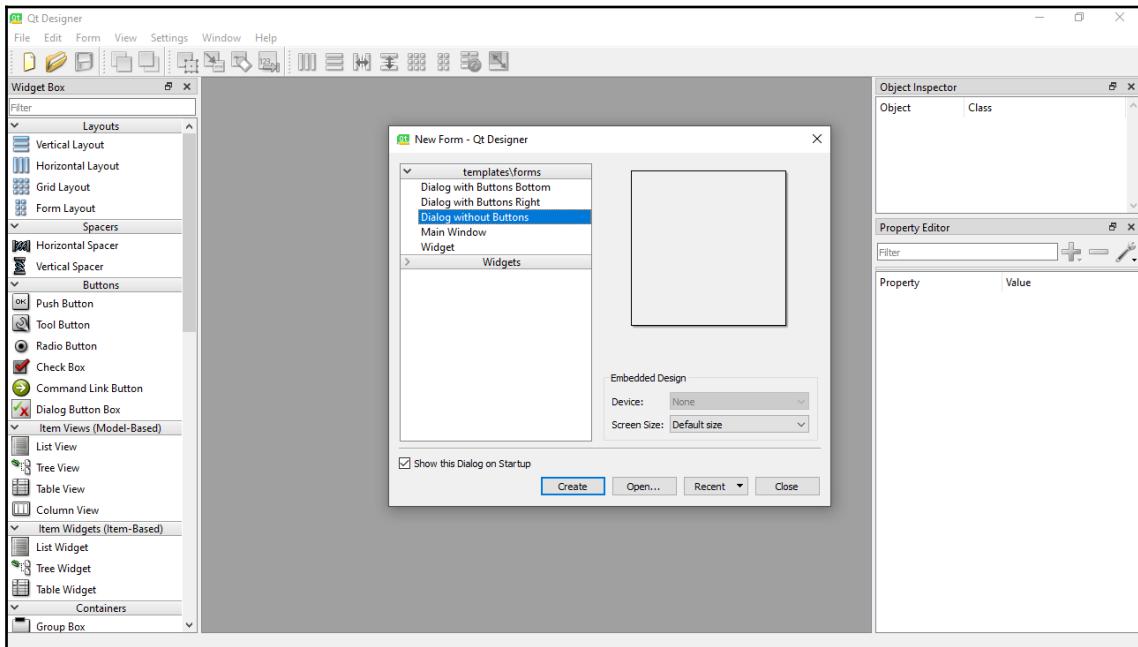
You will need to have both PyQt5 and the Qt Designer tool installed on your computer. Please read the *Installing PyQt5* and *Installing the PyQt5 Designer tool* recipes to find out how to do this.

How to do it...

You will need to run the `Designer.exe` file. Its location can be found in the *Installing the PyQt5 Designer tool* recipe.

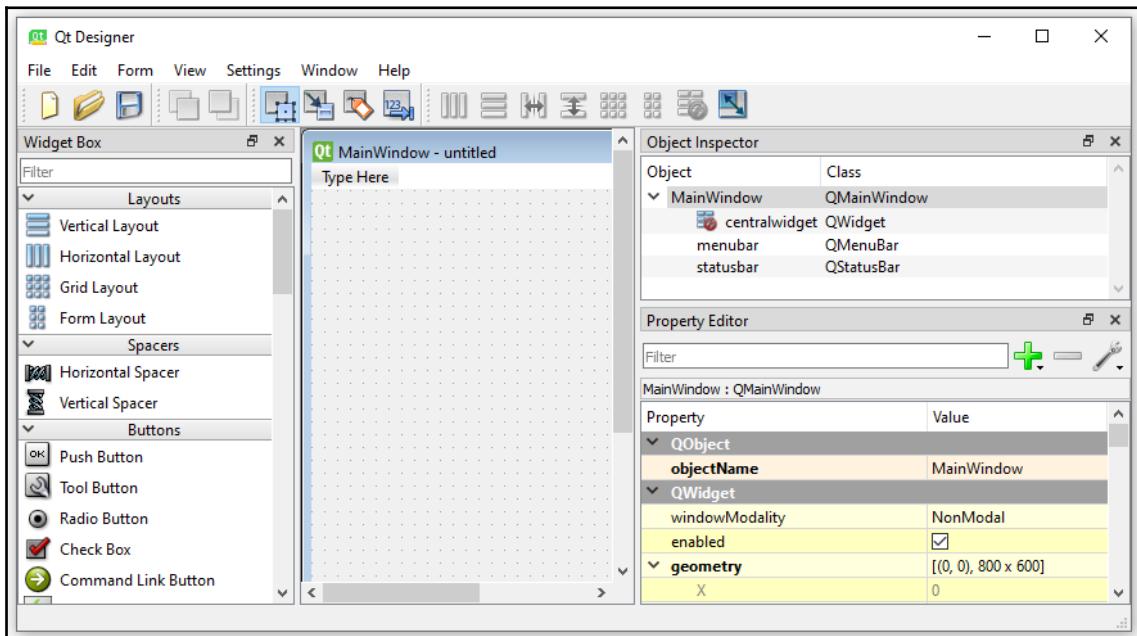
Let's get started:

1. Locate `Designer.exe` and double-click on it to run it.
2. The Designer GUI will open, as follows:



3. In the **New Form - Qt Designer** dialog box, as shown in the preceding screenshot, change the default in the top-left corner to **Main Window**.

4. Click the **Create** button in the dialog.
5. You should see the **Qt Designer** change to the following view:



Let's go behind the scenes to understand what we are seeing.

How it works...

In *step 1*, we are starting the **Qt Designer** by double-clicking the executable. In *step 2*, we can see that, by default, we are being presented with a dialog form that lets us create a new UI or open an existing UI.

The form directly behind the dialog box is dark grey, which means it is empty. This is actually the area in which we design our GUIs.

On the left-hand side, we can see the **Widget Box** area. This area contains all of the PyQt5 widgets the Designer has access to. We will drag and drop widgets from this **Widget Box** onto the UI form.

On the right-hand side of the Designer, we have two windows: the **Object Inspector** and the **Property Editor**. Both are currently empty.

In *step 3*, we change the default setting to **Main Window** because we want to create a `QMainWindow` application. In the previous recipes, we did this manually, but here we are using the Designer to do this for us.

In *step 4*, we click the **Create** button, which closes the dialog and creates a new **Main Window** form in the center area of the Designer. At the same time, the two windows on the right-hand side are no longer empty.

In *step 5*, we note that the classes and properties the **Main Window** has. In the **Object Inspector**, we can see four classes: `QMainWindow`, `QWidget`, `QMenuBar`, and `QStatusBar`. In the previous recipes, we manually added a menu bar and a status bar. Using the Designer tool when creating a new `QMainWindow`, we can see that the Designer has automatically added this functionality for us.

In the **Property Editor**, we can see the **geometry** property for the **centralwidget** object. This is a `QWidget` and is the central part of the entire **Main Window**. The **menubar** and **statusbar** are located above and below the central form, respectively. The values for the **geometry** property default to 800×600 , which will become the resultant size of our GUI when we run the code the way it is.



We can use this property to change the size of the UI. Alternatively, we can drag the UI form to the center of the Designer to change its size. This will update this property so that it works in both ways.

Take a look around the Designer to get a feel for how it works and what information it provides.

Previewing the form within the PyQt5 Designer

In this recipe, we will learn how to preview the form we are creating with the Designer. This is a very useful feature the Designer offers us because we can make changes, undo them, preview them, and so on until we are happy with our design. At that point, we can save the design.

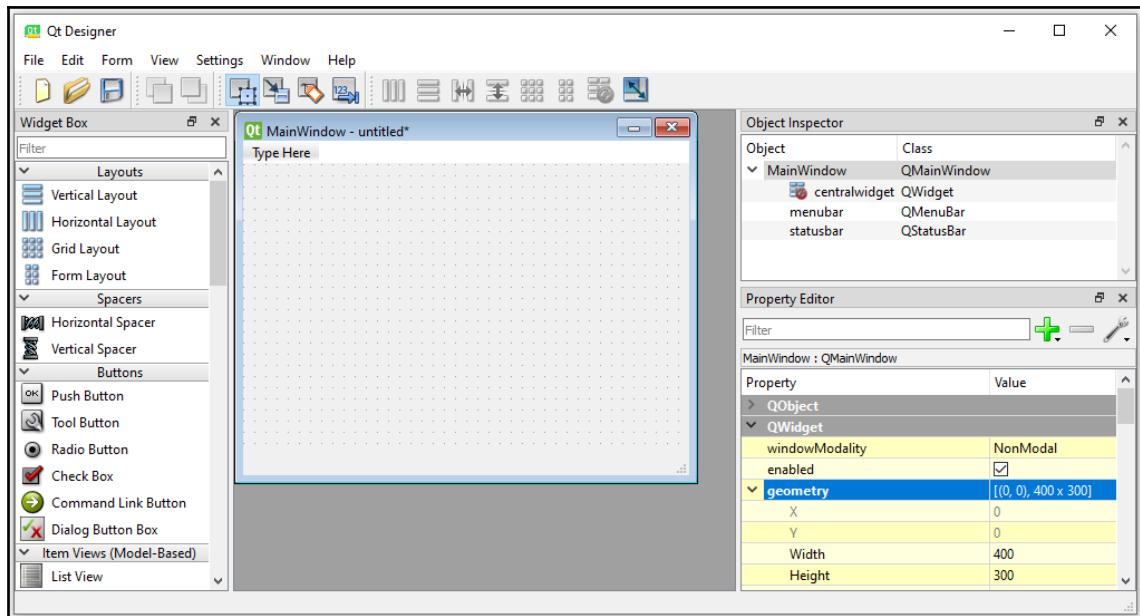
Getting ready

You will need to have both PyQt5 and the Qt Designer tool installed on your computer.

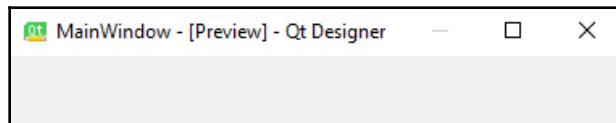
How to do it...

Run `Designer.exe`, as explained in the previous recipe. We will change the Main Window size and then preview it. Follow these steps to learn how to preview the form:

1. Perform *steps 1 to 5* from the previous recipe.
2. In the **Property Editor**, change the **geometry** property to 400×300 , as shown in the following screenshot:



3. In the Designer menu, click **Form | Preview...** or press *Ctrl + R*.
4. You should see the following preview:

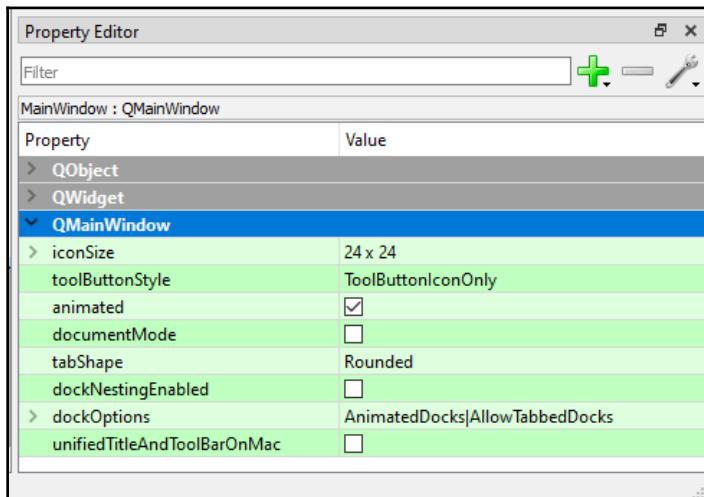


Let's go behind the scenes to understand this code better.

How it works...

In *step 1*, we are performing the same steps that we performed in the previous recipe. This brings us back to the same stage, because once we close the Designer tool, our UI will be lost if we do not save it. We haven't saved it so far.

In *step 2*, we are using the **Property Editor** on the right-hand side of the Designer to change the size of our UI. Make sure you have **QWidget** selected in this editor and not **QMainWindow**. If your editor looks like the following screenshot, simply expand the **QWidget** properties by clicking the arrow to the left of it:



In *step 3*, we are previewing our current UI design. There are two ways to do this: clicking the menu item and pressing the key shortcut.

Step 4 shows the resultant UI. Note the word **[Preview]** in the title bar of the window.

Saving the PyQt5 Designer form

In this recipe, we will add the same menu and menu item that we created previously. We will save our UI after previewing it.

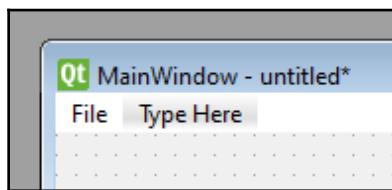
Getting ready

You will need to have both PyQt5 and the Qt Designer tool installed on your computer.

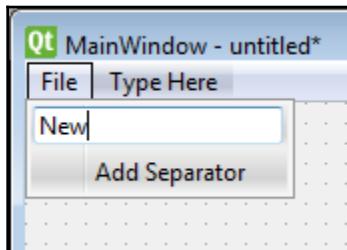
How to do it...

Run `Designer.exe`, as explained in the previous recipe. In order to create the menu and menu item, we can simply type into the Main Window within the Designer. Moving on, perform the following steps:

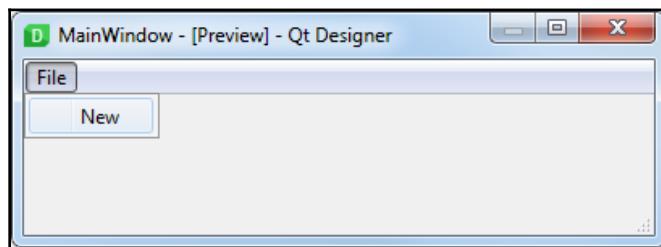
1. Perform *step 1* and *step 2* from the previous recipe.
2. In the Designer, inside **MainWindow - untitled***, type `File` into the **Type Here** menu, as shown in the following screenshot:



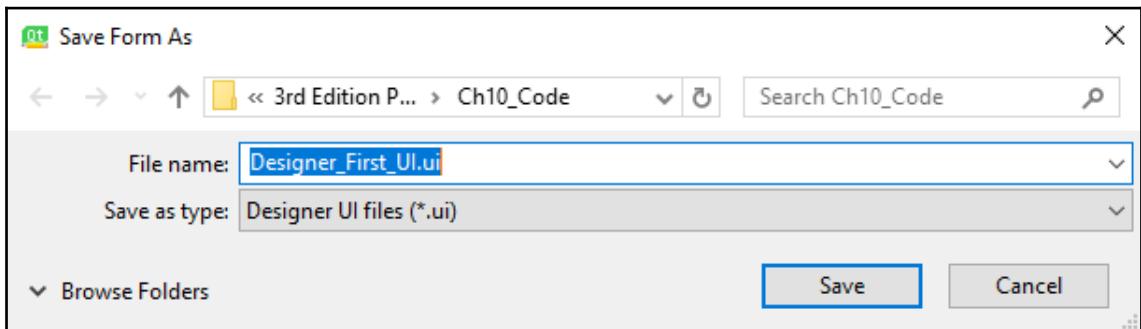
3. Click on **File**, type **New**, and press the *Enter* key to create a menu item:



4. Press *Ctrl + R* to preview the UI:



5. Close the preview and save the design in the Designer as `Designer_First_UI.ui`, as shown in the following screenshot:



Let's go behind the scenes so that we can understand these steps better.

How it works...

In *step 1*, we are performing the same steps that we performed in the previous recipe. In *step 2*, we are creating a **File** menu by simply typing into the menu bar the Designer has provided for us. In *step 3*, we add a menu item to this menu, also simply by typing into **Type Here** below our new menu.

In *step 4*, we preview our UI design, while in *step 5*, we are actually saving our design for the first time.

Note how the extension of UIs we design in the Designer is `.ui`.

Converting Designer `.ui` code into `.py` code

In this recipe, we will look at the `.ui` code we saved in the previous recipe when we saved our design in the Qt Designer tool. After that, we will use a utility we installed during the installation of the PyQt5 tools that will convert the `ui` code into Python `py` code.

We will be specifically using the `pyuic5` tool. You can think of the name as follows:

*Generate Python **py** code from the Designer **ui** code by converting it, using PyQt version 5.*



If you are trying to find where `pyuic5.exe` is located, it actually gets installed into the `Python\Scripts` subfolder. On my installation, this is `C:\Python37\Scripts\pyuic5.exe`. Make sure your `PATH` is set to the `Scripts` folder in order to successfully run it.

Let's get ready.

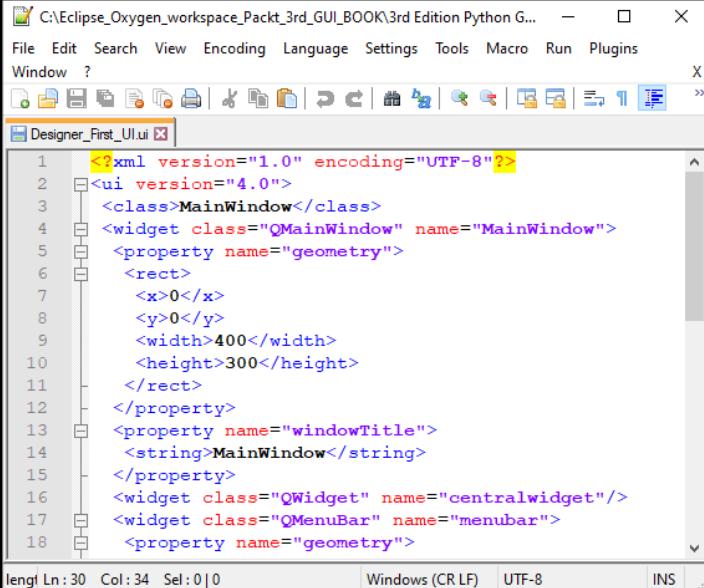
Getting ready

You will need to have the PyQt5 tools installed on your computer.

How to do it...

First, we will open the .ui code we generated in the previous recipe when we saved our UI in the Designer. Now, follow these steps:

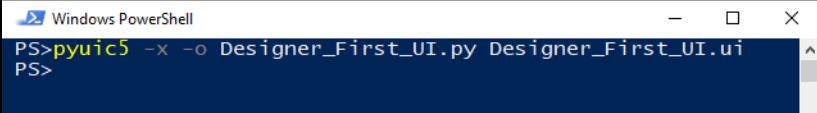
1. Open `Designer_First_UI.ui` from the previous recipe in a word editor such as Notepad++.
2. Look at the .ui code:



The screenshot shows the Notepad++ interface with the file `Designer_First_UI.ui` open. The code is an XML snippet representing a Qt UI file. It defines a `MainWindow` window with a width of 400 and height of 300 pixels. The window title is set to `MainWindow`. The central widget is a `QWidget`, and there is a menu bar with a single item named `menubar`.

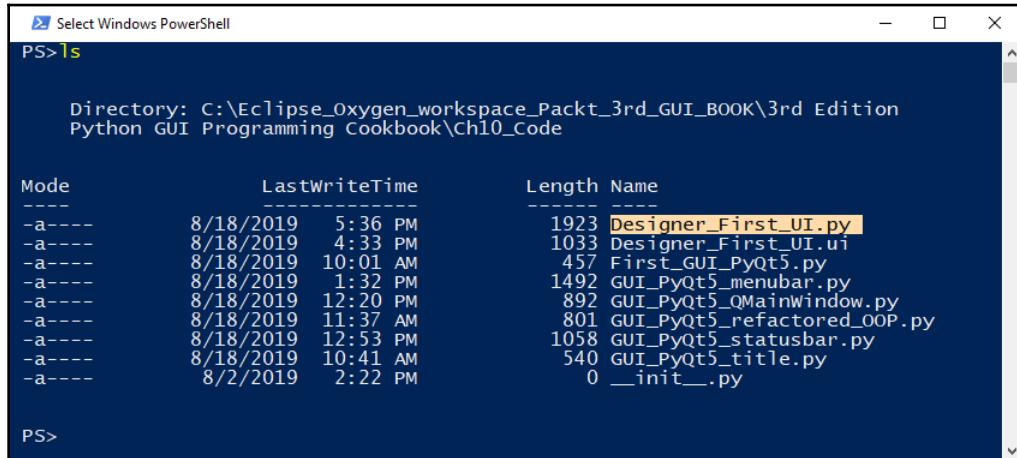
```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
<width>400</width>
<height>300</height>
</rect>
</property>
<property name="windowTitle">
<string>MainWindow</string>
</property>
<widget class="QWidget" name="centralwidget">
<widget class="QMenuBar" name="menubar">
<property name="geometry">
```

3. Navigate to the location on your hard drive where you saved `Designer_First_UI.ui` and open a Windows PowerShell or Command Prompt window.
4. Type in the `pyuic5 -x -o Designer_First_UI.py` `Designer_First_UI.ui` command and press the *Enter* key as shown in the following screenshot:



The screenshot shows a Windows PowerShell window with the command `PS>pyuic5 -x -o Designer_First_UI.py Designer_First_UI.ui` entered. The command is used to generate Python code from the UI file.

- Run the `ls` command in PowerShell or the `dir` command in a Command Prompt window to see the newly generated `.py` file. Alternatively, use Windows File Explorer to see the new file:



Mode	LastWriteTime	Length	Name
-a---	8/18/2019 5:36 PM	1923	Designer_First_UI.py
-a---	8/18/2019 4:33 PM	1033	Designer_First_UI.ui
-a---	8/18/2019 10:01 AM	457	First_GUI_PyQt5.py
-a---	8/18/2019 1:32 PM	1492	GUI_PyQt5_menuBar.py
-a---	8/18/2019 12:20 PM	892	GUI_PyQt5_QMainWindow.py
-a---	8/18/2019 11:37 AM	801	GUI_PyQt5_refactored_OOP.py
-a---	8/18/2019 12:53 PM	1058	GUI_PyQt5_statusBar.py
-a---	8/18/2019 10:41 AM	540	GUI_PyQt5_title.py
-a---	8/2/2019 2:22 PM	0	__init__.py

Let's go behind the scenes to understand these conversion steps better.

How it works...

In *steps 1 and 2*, we are opening the `.ui` file that got saved in the Designer. We are using Notepad++ or any other word editor for this. The resultant `.ui` output is clearly XML.

This is definitely not Python code. We have to convert the XML into Python code, which we do in *steps 3 and 4*.



The `-x` argument after `pyuic5.exe` makes the resultant Python module executable, while `-o` specifies the name of the output file. We have chosen the same name as the `.ui` file but with a `.py` extension. We can choose any name we wish, as long as it has a `.py` extension. The `pyuic5` utility also has the capability to convert more than one `.ui` file into a single `.py` file, so being able to choose a name comes in handy.

When running `pyuic5.exe`, we do not get any output if the conversion was successful. If we do not get any errors, this means the conversion was successful.

In *step 5*, we verify that we have the new output file, that is, `Designer_First_UI.py`.

Understanding the converted Designer code

In the previous recipe, we converted the Designer UI code into Python code using the `pyuic5` converter tool. In this recipe, we will look at the generated code. Every GUI we create with the Designer needs to be converted and any changes we make will overwrite all the previous code. This will allow us to understand how to decouple UI code from the functionality we will add to the UI using a modular approach in Python.

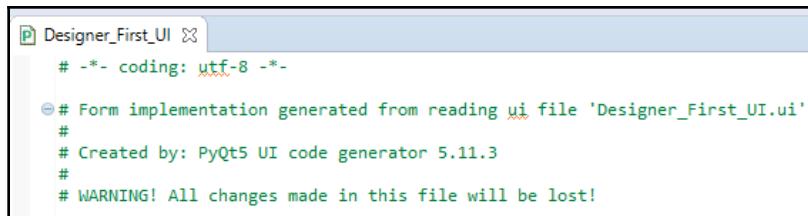
Getting ready

You will need to have the converted code from the previous recipe available. If you did not follow the preceding recipes in this chapter, simply download the necessary code from the Packt website for this book. The website provides all of the code for this book and you can simply click one button to download it all via GitHub.

How to do it...

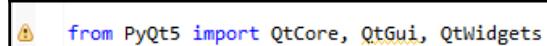
We will need to open the `.py` code we converted from the `.ui` code to understand its structure. Now that we've done this, we can follow these steps:

1. Open `Designer_First_UI.py` from the previous recipe.
2. Note the top section of the auto generated module:



```
Designer_First_UI.py
# -*- coding: utf-8 -*-
# Form implementation generated from reading ui file 'Designer_First_UI.ui'
#
# Created by: PyQt5 UI code generator 5.11.3
#
# WARNING! All changes made in this file will be lost!
```

3. Look at the import statements just below the preceding section:



4. Look at the class that was created and its first method:

```
class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(400, 300)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setObjectName("menubar")
        self.menuFile = QtWidgets.QMenu(self.menubar)
        self.menuFile.setObjectName("menuFile")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)
        self.actionNew = QtWidgets.QAction(MainWindow)
        self.actionNew.setObjectName("actionNew")
        self.menuFile.addAction(self.actionNew)
        self.menubar.addAction(self.menuFile.menuAction())

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)
```

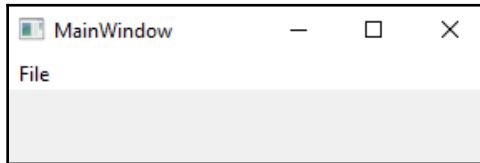
5. Look at the second method, which is below the first method within the class:

```
def retranslateUi(self, MainWindow):
    _translate = QtCore.QCoreApplication.translate
    MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
    self.menuFile.setTitle(_translate("MainWindow", "File"))
    self.actionNew.setText(_translate("MainWindow", "New"))
```

6. Lastly, look at the "`__main__`" section at the bottom of the code:

```
if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

7. Run this code. The result should be a running Python GUI:



Let's go behind the scenes to better understand the code.

How it works...

In *step 1*, we open the converted UI as a Python module. In *step 2*, we can see an important warning.



I strongly recommend that you take this warning **seriously**. If you add code to this module and regenerate the code via `pyuic5.exe` at a later date, all of your changes will indeed be **lost**!

Step 3 shows us the three import statements. These are always being imported, although `QtGui` is not required, as can be seen by the yellow warning and underline in my Eclipse/PyDev editor.

Step 4 shows us the class that's always created. It is immediately followed by the `setupUi` method. There is no `__init__` method in between. The code in this method is very important for us because we can access the class attributes via the generated names.

In *step 5*, we note the `retranslateUi` method. This method is also auto generated. By taking a closer look, we can find the names of the menu and menu items we added during the UI design phase.

Step 6 shows us the "`__main__`" section at the bottom of the code. The important thing to know about this is that this section is only created when we specify the `-x` option during the `pyuic5` conversion. If we leave this option out, we won't see this section.

In *step 7*, we run our GUI. Note how we are no longer *previewing* the UI. This is real and pure Python code now.

Building a modular GUI design

As we saw in the previous recipe, all of the auto generated code of the UI we are designing with the Designer will be overwritten as soon as we rerun the `pyuic5` utility. This is a good thing because it encourages us to design our Python modules in a modular fashion (hence the name *module*).

In this recipe, we will import the generated UI from a new Python module and add functionality within it. Whenever we rerun the `pyuic5` utility, our code will not get accidentally overwritten, because we are separating the logic from the UI.



Separation of Concerns (SoC) is a software term that refers to the benefits of good, modular design.

So, let's write some code!

Getting ready

You will need the converted code from the previous recipe, which can be found in `Designer_First_UI.py`.

How to do it...

We will create a new module in which we will add functionality to our UI code. We will import the UI we created in the Qt Designer that we converted into Python code. Let's get started:

1. Create a new Python module and name it `Designer_GUI_modular.py`.
2. In this module, write the following line of code:

```
from Ch10_Code.Designer_First_UI import Ui_MainWindow
```



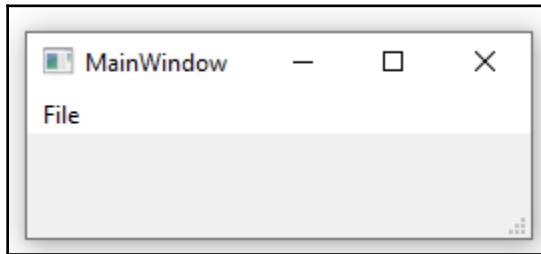
Note that you might have to adjust the `Ch10_Code` prefix to match your location.

3. Run the preceding code. You shouldn't get any errors.
4. Next, copy the "`__main__`" section from `Designer_First_UI.py` into this new module.
5. You will also need to import `QtWidgets` to make this work:

```
from PyQt5 import QtWidgets
from Ch10_Code.Designer_First_UI import Ui_MainWindow

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

6. Now, run the preceding code. You should see the GUI we designed and previously ran stand-alone:



Let's go behind the scenes to better understand the code.

How it works...

In *step 1*, we create a new Python module. In *step 2*, all we do is import the UI into the Python generated code.

This is very important, because it shows the principle of SoC!



In *step 3*, we run this one line of code. No GUI will be visible, but the important thing here is that we don't get any errors. If we get some errors, it typically means that our import statement failed because our module could not locate the module we are trying to import.

Step 4 copies the "`__main__`" section from our converted `.py` file. While the module runs by itself, when we import it, we also have to import `QtWidgets`, because when we import modules, the import statements of those modules do not automatically get imported. We do this in *step 5*. In *step 6*, we have our GUI up and running, but this time via a modular approach.

Adding another menu item to our menu bar

In this recipe, we will add a second menu item to our GUI. We will use the Designer and then regenerate the UI code. After that, we will attach functionality to the menu item from our modular Python module. The Designer has certain capabilities so that it can add this functionality as well, but here, we are simply keeping the UI code separated from the functionality of our GUI.

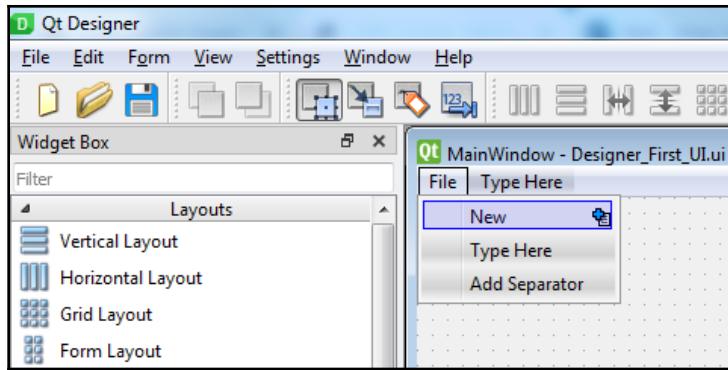
Getting ready

You will need to have the UI code from the previous recipes available. All the other recipes' prerequisites apply to this recipe as well.

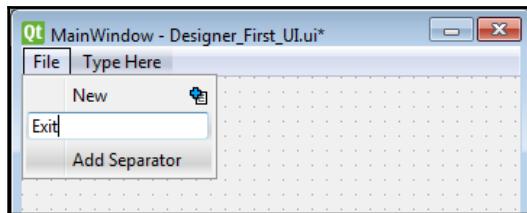
How to do it...

We will enhance our UI design from a previous recipe by adding a second menu item. After that, we will convert the UI code into Python code, like we did previously. Let's get started:

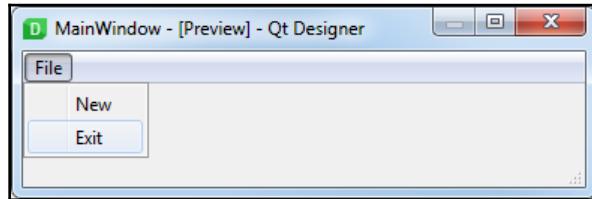
1. In the Qt Designer, open `Designer_First_UI.ui`.
2. Below the `File | New` menu item, create another menu item and name it `Exit`:



Type this new menu item into the **Type Here** area. It will look like this:



3. Press the *Enter* key and save the .ui file. Next, preview the UI:



4. Run the `pyuic5.exe` utility to convert the .ui file into a .py file. Let's save it under a new name to distinguish it from our original module.

In a PowerShell or Command Prompt window, type `pyuic5.exe -x -o Designer_First_UI_Exit.py Designer_First_UI.ui` and then press the *Enter* key:

```
PS>pyuic5.exe -x -o Designer_First_UI_Exit.py Designer_First_UI.ui
PS>
```

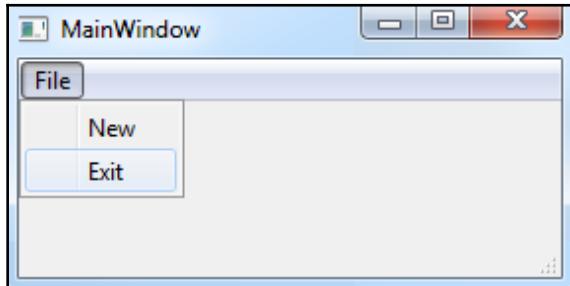
You should now have a new Python module named `Designer_First_UI_Exit.py`.

5. Create a new Python module and name it `Designer_GUI_modular_exit.py`. Import the newly converted file into it. Here is what the code looks like:

```
from PyQt5 import QtWidgets
from Ch10_Code.Designer_First_UI_Exit import Ui_MainWindow

if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
    MainWindow.show()
    sys.exit(app.exec_())
```

6. Run the GUI and click on the **File** menu to see the new **Exit** menu item. The result will be the same one we got in *step 3* when we previewed the running GUI in the Designer before we converted it:



Now, let's go behind the scenes to understand what's going on.

How it works...

In *step 1*, we are opening the `.ui` design file we created previously. We aren't saving it under a different name, so we are basically adding a new menu item to our existing design.

In *step 2*, we use the Designer tool to add a new menu item and name it `Exit`.

In *step 3*, we press the `Enter` key, which, quite honestly, sounds very trivial, but if we don't do this, our new item will not be saved. We also save the `.ui` file under the same name, overwriting our previous version. This is okay because we are simply adding some small functionality to the UI.

In *step 4*, we are running the `pyuic5.exe` utility to turn the XML of the `.ui` file into Python code. This time, however, we are giving the resultant `.py` output file a different name than the `.ui` file. We do this so that we don't overwrite our previous Python module.

In *step 5*, we create a new Python module and import the converted `.ui` into it. We have done this before. Finally, in *step 6*, we run the pure Python code so that we can see our new menu item.

There's more...

In the next recipe, we will add functionality to our new `Exit` menu item so that when we click on it, our GUI will indeed exit and the application will end.

Connecting functionality to the Exit menu item

In this recipe, we will add functionality to the `Exit` menu item we created in the previous recipe. So far, we have two menu items, but they aren't interactive.

Here, we will learn how to add functionality outside of the UI by using our modular approach to coding. We will also improve our code by transforming the "`__main__`" self-testing section into a class of its own.

Getting ready

You will need to have the .ui code from the previous recipe available. All the other prerequisites apply to this recipe as well.

How to do it...

We will add functionality to our GUI using the modular approach of SoC. To make our code more robust, we will create a new class. Let's get started:

1. Create a new Python module and name it

Designer_GUI_modular_exit_class.py.

2. Type the following import statements into the module:

```
import sys
from PyQt5 import QtWidgets
from Ch10_Code.Designer_First_UI_Exit import Ui_MainWindow
```

3. Create a new Python class with an `__init__` method:

```
class ExitDesignerGUI():
    def __init__(self):
        app = QtWidgets.QApplication(sys.argv)
        self.MainWindow = QtWidgets.QMainWindow()
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self.MainWindow)
        self.update_widgets()
        self.widget_actions()
        self.MainWindow.show()
        sys.exit(app.exec_())
```

4. Create a method that updates the status bar and connects an action to the menu item:

```
def widget_actions(self):
    self.ui.actionExit.setStatusTip(
        'Click to exit the application')
    self.ui.actionExit.triggered.connect(self.close_GUI)
```

5. Write a callback method that closes the GUI:

```
def close_GUI(self):
    self.MainWindow.close()
```

6. Write a class method that updates the title of the GUI:

```
def update_widgets(self):  
    self.MainWindow.setWindowTitle('PyQt5 GUI')
```

7. Create an instance of the class in the "`__main__`" section:

```
if __name__ == "__main__":  
    ExitDesignerGUI()
```

Now, let's see how this works.

How it works...

In *step 1*, we are creating a new Python module, while in *step 2*, we are writing the import statements we need at the top of the module.



Writing all the import statements at the top of a module is a Python best practice and is highly recommended.

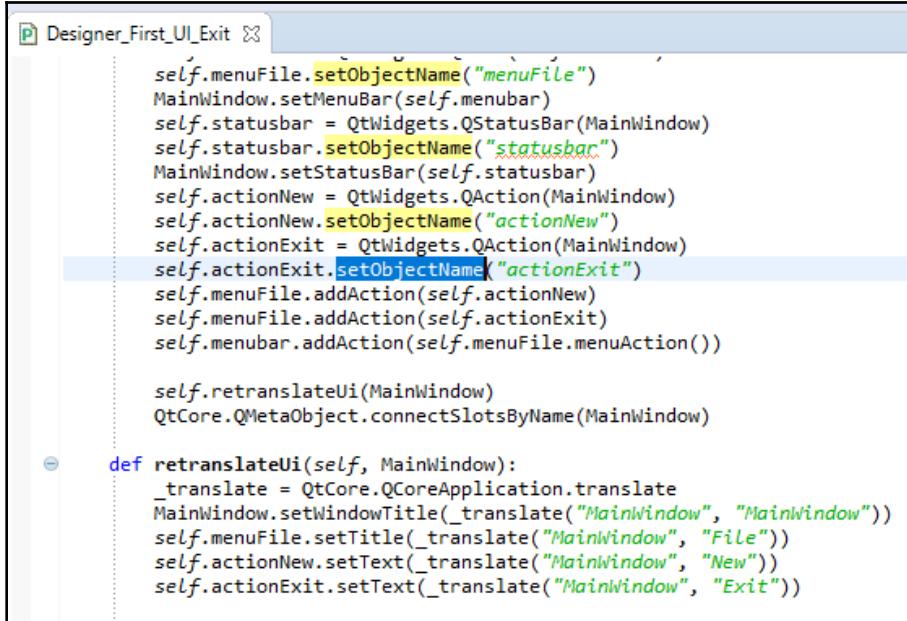
In *step 3*, we are creating our own Python class and starting with the typical initializer. Toward the end of the initializer method, we are calling methods that we are creating below the initializer. Because we are doing it like this, all we have to do is instantiate the class, and the methods will run without us having to call any particular methods on the class instance after the creation of the class instance (object).

In *step 4*, we are accessing the name of our menu item via `self.ui.actionExit`. We can use `self.ui` because, in the "`__init__`" method, we created an instance of the `MainWindow` and saved it as such. This is the line of code:

```
self.ui = Ui_MainWindow()
```

How come we use `actionExit`? We have to look at the auto generated code to find this object name.

We designed our UI in the Designer and the Designer chose a name for us. We could change that object name if we wish, but it isn't necessary. We just have to find the name we are looking for. So, let's look at `Designer_First_UI_Exit.py`:



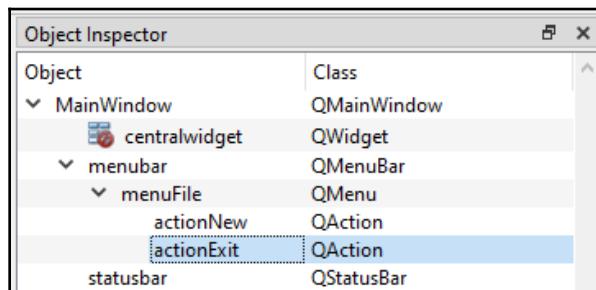
```
self.menuFile.setObjectName("menuFile")
MainWindow.setMenuBar(self.menubar)
self.statusbar = QtWidgets.QStatusBar(MainWindow)
self.statusbar.setObjectName("statusbar")
MainWindow.setStatusBar(self.statusbar)
self.actionNew = QtWidgets.QAction(MainWindow)
self.actionNew.setObjectName("actionNew")
self.actionExit = QtWidgets.QAction(MainWindow)
self.actionExit.setObjectName("actionExit")
self.menuFile.addAction(self.actionNew)
self.menuFile.addAction(self.actionExit)
self.menubar.addAction(self.menuFile.menuAction())

self.retranslateUi(MainWindow)
QtCore.QMetaObject.connectSlotsByName(MainWindow)

def retranslateUi(self, MainWindow):
    _translate = QtCore.QCoreApplication.translate
    MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
    self.menuFile.setTitle(_translate("MainWindow", "File"))
    self.actionNew.setText(_translate("MainWindow", "New"))
    self.actionExit.setText(_translate("MainWindow", "Exit"))
```

We can find the "actionExit" name in this file.

We can also look it up in **Qt Designer** in the **Object Inspector**:



Either way works.

Also in *step 4*, we are achieving the desired functionality, that is, closing our GUI via the **Exit** menu item.

We do this in the following line of code, which calls the method we create in *step 5*:

```
self.ui.actionExit.triggered.connect(self.close_GUI)
```

The very important thing to note is that we call `triggered.connect (<method name>)` to achieve this functionality. This connects the *action* to the event of the menu item being *triggered*.



This is PyQt5 syntax and semantics.

In *step 5*, in order to close the GUI, we call the built-in `close` method on `MainWindow`. We saved a reference to `MainWindow` in the "`__init__`" method so that we can reference it from within this method, within the same Python class.

In *step 6*, we use the same reference of `self.MainWindow` to give our GUI window a title.

In *step 7*, we use Python's self-testing "`__main__`" section to create a class instance. This is all we need to do to run our GUI.

Now, when we click on the `Exit` menu item, our GUI will exit.

Adding a Tab Widget via the Designer

In this recipe, we will add a Tab Widget to our UI using the Designer tool. Then, we will convert the `.ui` code into Python code. This will serve us well in preparation for adding more widgets and functionality to our GUI.

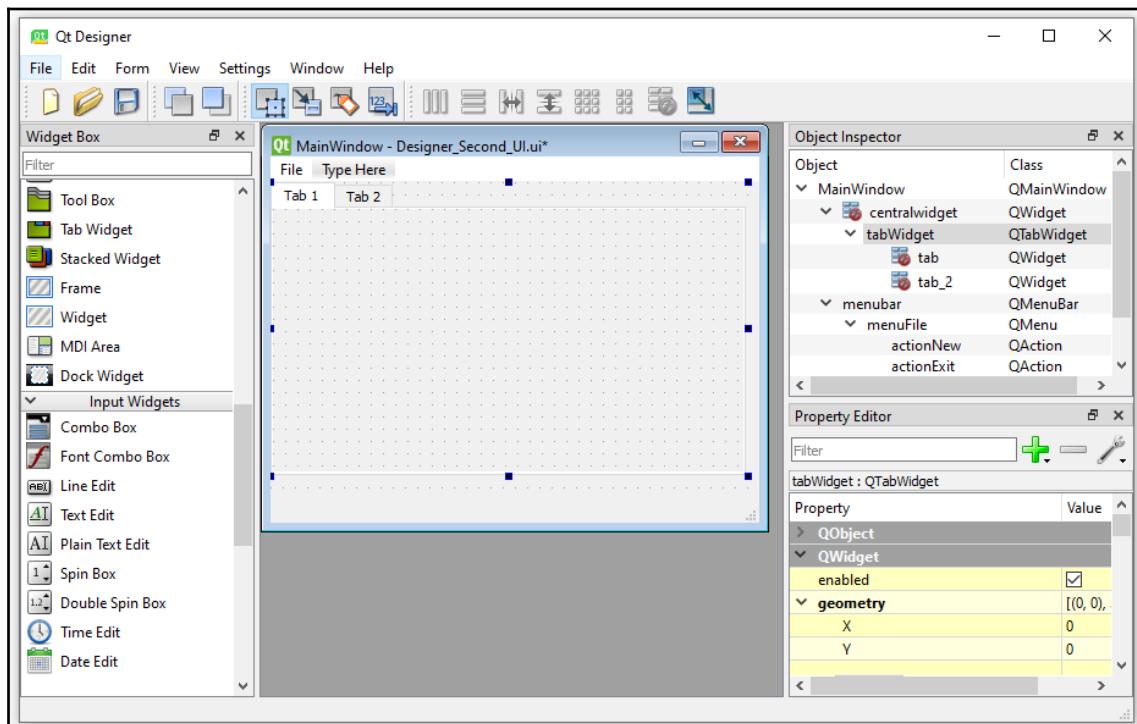
Getting ready

You will need to have the `.ui` code from the previous recipe available. All the other prerequisites apply to this recipe as well.

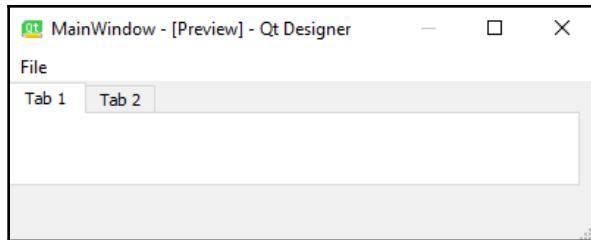
How to do it...

We will add a Tab Widget to our UI design by using the Designer. It is as simple as dragging and dropping. Let's get started:

1. In Qt Designer, open `Designer_First_UI.ui` and save it as `Designer_Second_UI.ui`.
2. From the left-hand side within the Designer, drag a **Tab Widget** onto the main form:



3. Resize the **Tab Widget** to make it fill up most of the `MainWindow`, as shown in the preceding screenshot.
4. Save the `.ui` design and preview it:



5. Click on **Tab 1** and **Tab 2**.
6. Click on the **Exit** menu item.
7. Use the `pyuic5.exe` tool to convert the `.ui` code into Python code.
8. Run the converted Python UI code.

How it works...

In *step 1*, we are opening our existing `.ui` file in the Qt Designer, but this time, we are saving it under a different filename. Saving the different versions of our UI design under different names is typically a good idea go back to prior versions if our UI design messes up.



Our nice UI design being accidentally messed up can happen quite easily, so make sure you make backups of your `.ui` files. You can also use a version control system such as GitHub to back up your code.

In *step 2*, we are using the fantastic drag and drop capability of the Designer to visually (and physically) move a widget onto the canvas form. We can simply use the resize handles to adjust the widget any way we want. We do this in *step 3*.



Look at the Object Inspector on the right-hand side of the Designer and note the new **Tab Widget**, as well as the two tabs, object names, and PyQt5 classes they belong to.

You may also notice a star (*) to the right of the `MainWindow` title. This means that we haven't saved the design yet. Pay attention to this because, if you close the UI design without saving it, you will lose your beautiful design.

In *step 4*, we save our UI design and preview it.

In *step 5*, we can click between the two new tabs. Note how the color of each tab defaults to white and looks different in preview mode than it does in design mode.

Step 6 will *not* close the UI because the functionality for closing the UI is decoupled from the UI. We wrote the code so that we could close the GUI in a different Python module.

Step 7 is mainly left as an exercise for you.



Get used to making small changes in the Designer, converting the code, importing it, running it, making more changes in the Designer, and so on. Get used to this rhythm of GUI development when you use the Designer tool.

In *step 8*, we run the code. Using the `-x` option creates the self-testing "`__main__`" section, so you should be able to run the converted Python code without having to import it.

Using layouts in the Designer

In this recipe, we will explore the very important concept of using layouts with PyQt5 and we will do so using the Designer tool. In `tkinter`, we explored Label Frames. In PyQt5, horizontal and vertical layouts are the main ways in which we can design our UI.



This can get a little bit tricky, so, as I mentioned previously, make sure you back up your UI design often so that, when things turn out ugly, you have a foundation to go back to.

In the following recipe, we will place widgets into these layouts.

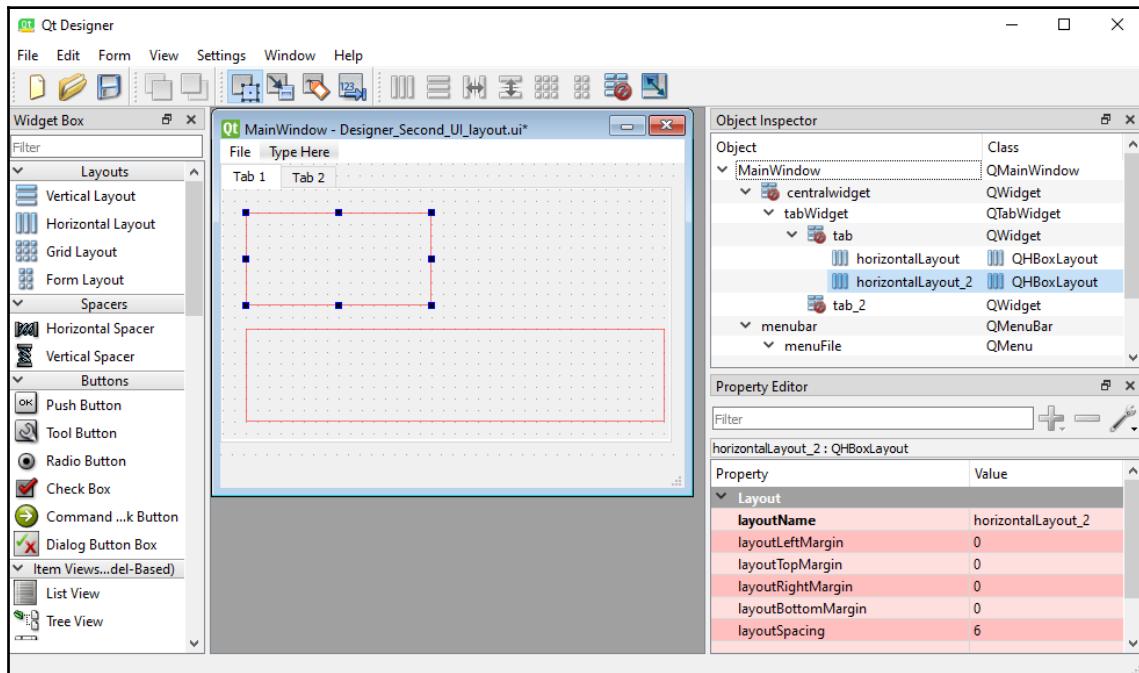
Getting ready

You will need to have the `.ui` code from the previous recipe available. All the other prerequisites apply to this recipe as well.

How to do it...

We will add two horizontal layouts to our Main Window, which we will do using drag and drop inside the Qt Designer. Let's get started:

1. In the Designer, open `Designer_Second_UI.ui` and save it as `Designer_Second_UI_layout.ui`.
2. Drag one horizontal layout toward the bottom of the form and resize it.
3. Drag a second horizontal layout and place it above the first layout:



Your `MainWindow` will now look like the preceding screenshot.

4. Save the design, convert it into Python, and then run it to make sure you do not get any errors. The code should run, but note that you won't see any difference in the resultant GUI.

How it works...

In *step 1*, we are saving our UI design under a different name, practically making a backup.

In *steps 2* and *3*, we are visually dragging horizontal layouts onto the main form. You can see them in the preceding screenshot, but you won't notice any difference in preview mode.

One very important thing to note is the names and classes in the **Object Inspector** in the top-right corner of the Designer, as well the properties that are automatically made available to us in the **Property Editor** in the bottom-right corner of the Designer.

Also, note how we are placing these widgets in **Tab 1**. **Tab 2** is still empty.



What's really cool about using these PyQt5 layouts is that, once we place widgets into these layouts, we can move the entire group of widgets by simply moving the layout. We can even hide them all, making them invisible!

In *step 4*, we are making sure that we know the process of making changes in the Designer and then converting it into Python code. If running the code doesn't work, something might have gone wrong. It's good to make small changes and then test your code.

Adding buttons and labels in the Designer

In this recipe, we will add a button and a label, both of which we will place in the layouts we added to our UI design in the previous recipe.

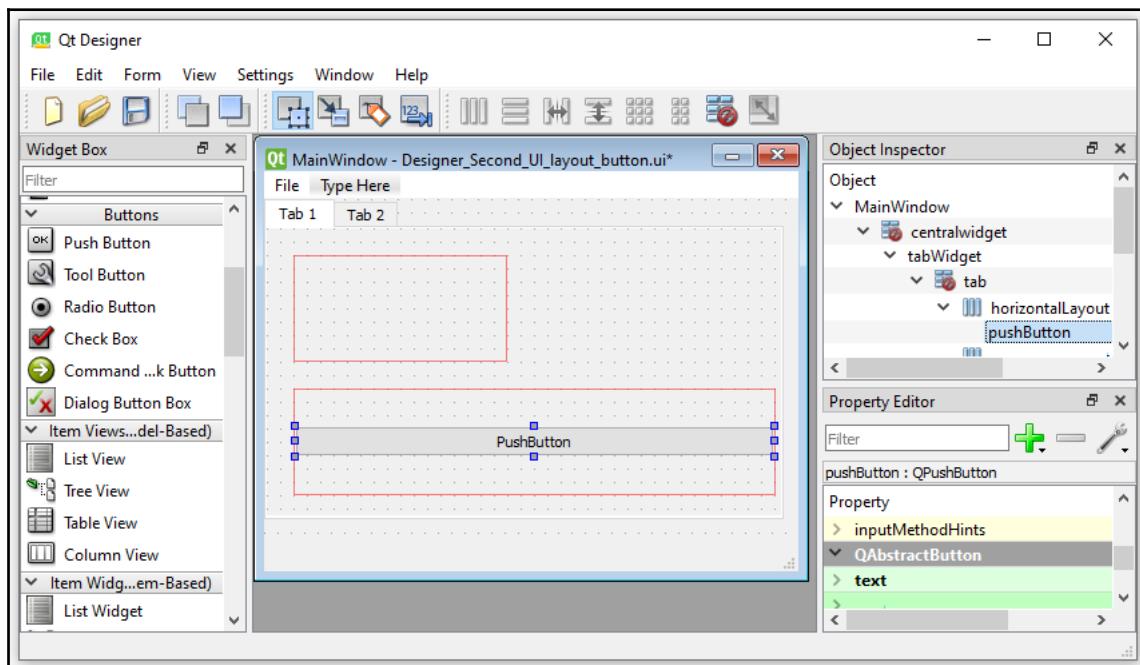
Getting ready

You will need to have the `.ui` code from the previous recipe available. All the other prerequisites apply to this recipe as well.

How to do it...

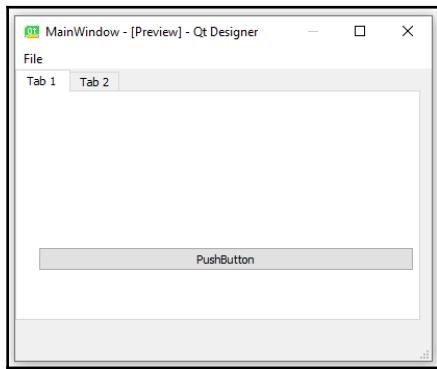
We will add a button and a label to our UI design. We will also use the Designer to connect the two, creating some of the available functionality directly within the Designer. Let's take a look at the steps:

1. In the Designer, open `Designer_Second_UI_layout.ui` and save it as `Designer_Second_UI_layout_button.ui`.
2. Drag a PushButton from the left-hand side into the lower horizontal layout:



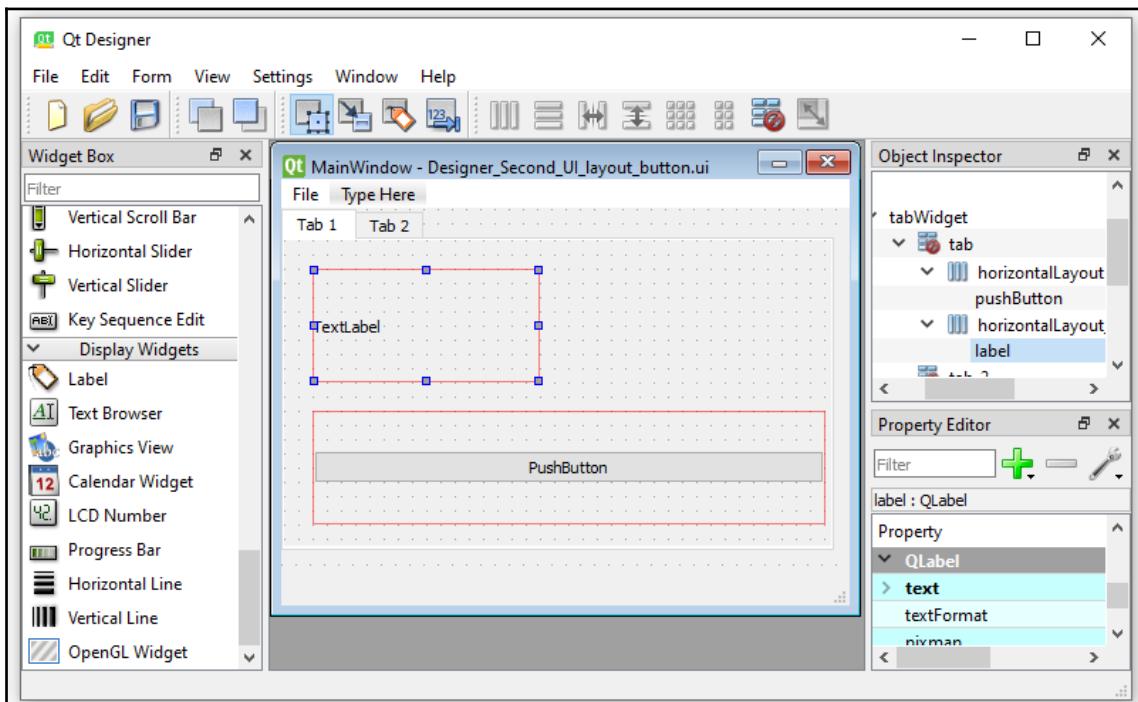
Note how the button automatically adjusts itself to the left and right edges of the horizontal layout.

3. Save and preview the UI. Click the button during the preview run of the UI:

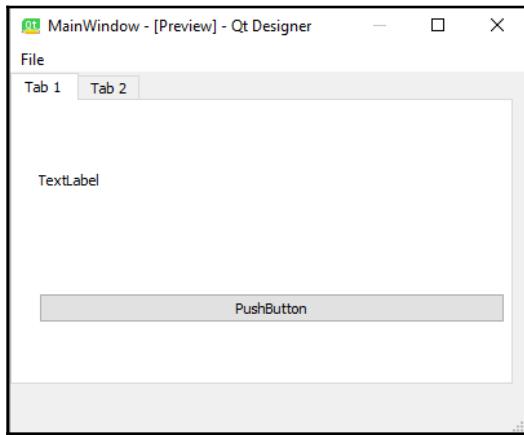


When clicking it, the button looks pressed in and comes with a default name we can change.

4. Drag a **Label** widget from the left-hand side and place it into the top horizontal layout:

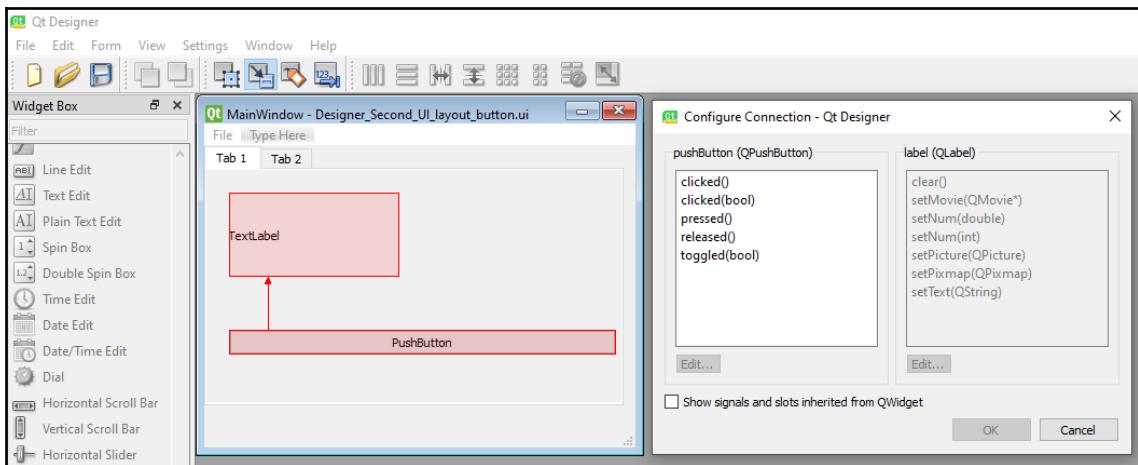


5. Save the .ui design and preview it:

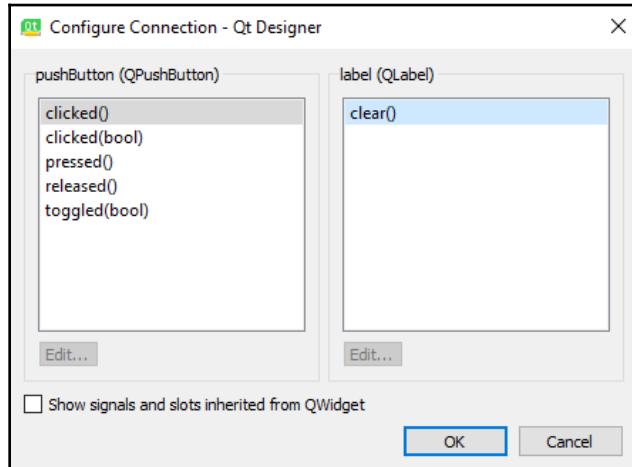


6. Next, we will connect the `PushButton` to the label within the Designer.

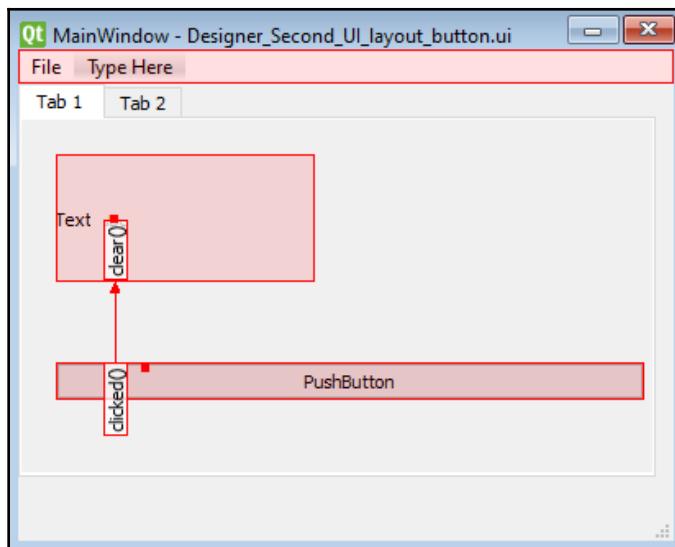
Press the *F4* key to get into the signals/slots editing mode or use the **Edit** menu within the Designer. Drag a signal/slot connection from the button to the label. The Designer will now look like this:



7. In the **Configure Connection** pop-up dialog, click on `clicked()` in the top-left corner. This will enable the right-hand side of the dialog box. Select the `clear()` method:



8. Press **OK** and save the `.ui` file. You will now see a connection between the button and the label:



9. Preview the `.ui` code and click the button. You will notice that the label text is cleared.
10. Convert the `.ui` code into Python code using `pyuic5.exe`, as we did previously.

Let's look at the converted code to understand how this functionality of connecting the button to clear the label works.

How it works...

In *step 1*, we are saving the UI design under a new name, while in *step 2*, we are dragging and dropping a `PushButton` onto the `MainWindow` form. By itself, this button doesn't do anything, but there are a few methods that are available within the Designer that can add functionality to it.

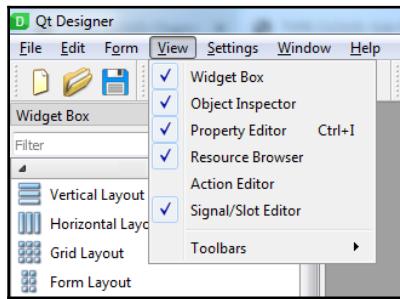
In *step 3*, we are previewing the UI. Note that the button was automatically stretched to fill in the entire horizontal layout box. Vertically, it is only as tall as it needs to be to display its default text. These are properties that we can change in the **Property Editor**.

In *step 4*, we are visually dragging a text label widget onto the `MainWindow`. This widget doesn't attach itself to the left- and right-hand sides of the horizontal layout, because it comes with a predefined width.

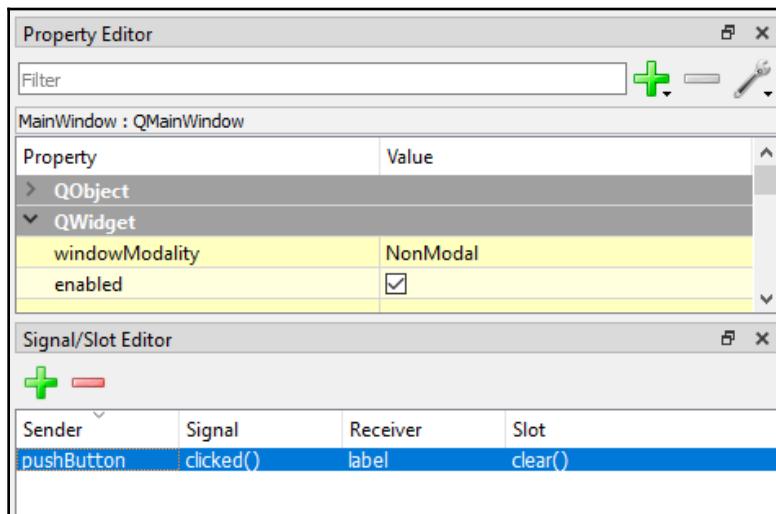
It attaches itself to the left-hand side. In the **Property Editor**, we can change this if we wish to.

Step 5 gives us a preview of our UI running from within the Designer.

Step 6 brings us into the **Signal/Slot Editor**. Signals and slots are peculiar to PyQt5. The following screenshot shows you how to enable this editor:



You can find the editor under the **View** menu. This opens a new window to the right, below the **Property Editor**:



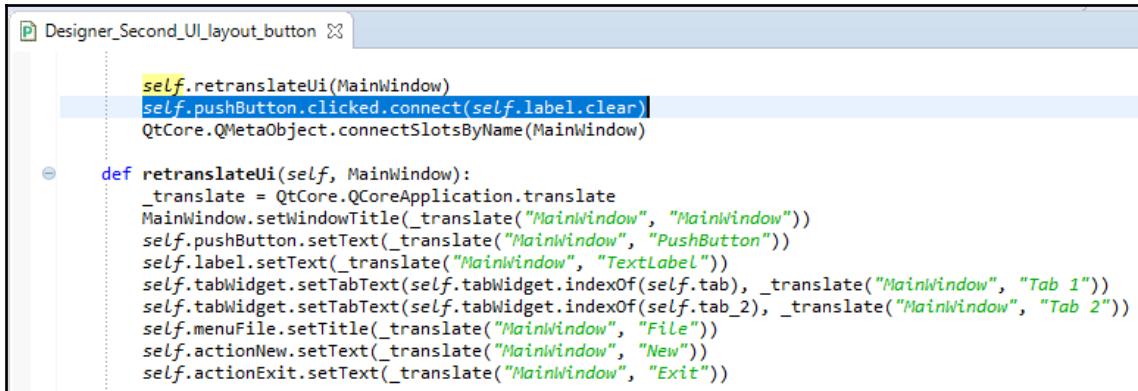
You can see that the `pushButton` object is the **Sender**, the **Signal** is `clicked()`, the **Receiver** is `label`, and the **Slot** is `clear()`.

Step 7 shows us some of the functionality that becomes available when we connect signals and slots between QPushButton and the QLabel widget. The `clear()` method is built in, so we can simply select it to clear the label whenever we push the button.

In step 8, we can see that the editor changed after we clicked **OK** to connect the two widgets.

Because the functionality has been added within the Designer, when we preview our UI, it actually works. Clicking the button does indeed clear the label, as can be seen in *step 9*.

When we convert our `.ui` code into Python code, we get the following output:



The screenshot shows the PyQt5 Designer interface with a window titled "Designer_Second_UI_layout_button". The code editor displays the generated Python code for the UI. The code includes the retranslateUi method and a connection for the pushButton.clicked signal.

```
self.retranslateUi(MainWindow)
self.pushButton.clicked.connect(self.label.clear)
QtCore.QMetaObject.connectSlotsByName(MainWindow)

def retranslateUi(self, MainWindow):
    _translate = QtCore.QCoreApplication.translate
    MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
    self.pushButton.setText(_translate("MainWindow", "PushButton"))
    self.label.setText(_translate("MainWindow", "TextLabel"))
    self.tabWidget.setTabText(self.tabWidget.indexOf(self.tab), _translate("MainWindow", "Tab 1"))
    self.tabWidget.setTabText(self.tabWidget.indexOf(self.tab_2), _translate("MainWindow", "Tab 2"))
    self.menuFile.setTitle(_translate("MainWindow", "File"))
    self.actionNew.setText(_translate("MainWindow", "New"))
    self.actionExit.setText(_translate("MainWindow", "Exit"))
```

Note how the `pushButton` has the same syntax of `clicked.connect (<method>)`, which is what we had outside of the Designer when we enabled the Exit menu item.

There's more...

The PyQt5 GUI framework is a very exciting tool to work with. I especially enjoy the Qt Designer tool.

Together with Packt, I have created several *Python GUI Programming Recipes using PyQt5* video courses.

One course is especially focused on PyQt5 GUI development. It is a little over four hours long.

Here's a screenshot of my course, which can be found at <https://www.packtpub.com/application-development/python-gui-programming-recipes-using-pyqt5-video>:



If this is too long for you (I am covering a lot of PyQt5 material in this course), there is also a shorter course, which focuses on tkinter and PyQt5, which can be found at <https://www.packtpub.com/application-development/hands-python-3x-gui-programming-video>:



I wish you the best of luck in your software development efforts.

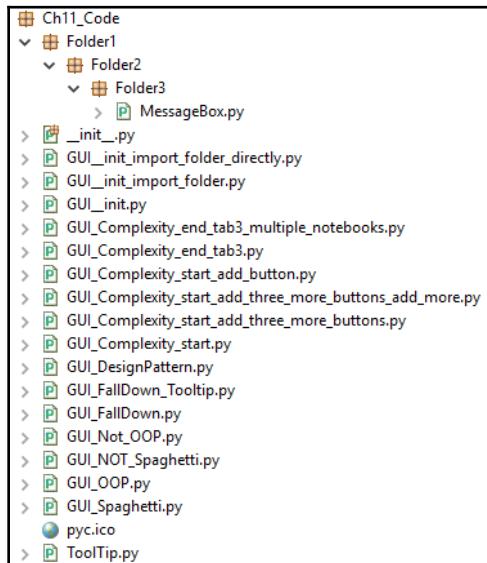
Python is a wonderful programming language.

11 Best Practices

In this chapter, we will explore the different *best practices* that can help us to build our GUI efficiently and keep it both *maintainable* and *extendable*.

These best practices will also help you to debug your GUI to get it just the way you want it to be.

Here is the overview of Python modules for this chapter:



Knowing how to code using best practices will greatly enhance your Python programming skills.

The recipes that will be discussed in this chapter are the following:

- Avoiding spaghetti code
- Using `__init__` to connect modules
- Mixing fall-down and OOP coding
- Using a code naming convention
- When not to use OOP
- How to use design patterns successfully
- Avoiding complexity
- GUI design using multiple notebooks

Avoiding spaghetti code

In this recipe, we will explore a typical way to create spaghetti code and then we will see a much better way of how to avoid such code.



Spaghetti code is code in which a lot of functionality is intertwined.

Getting ready

We will create a new, simple GUI, written in Python using the built-in Python `tinkter` library.

How to do it...

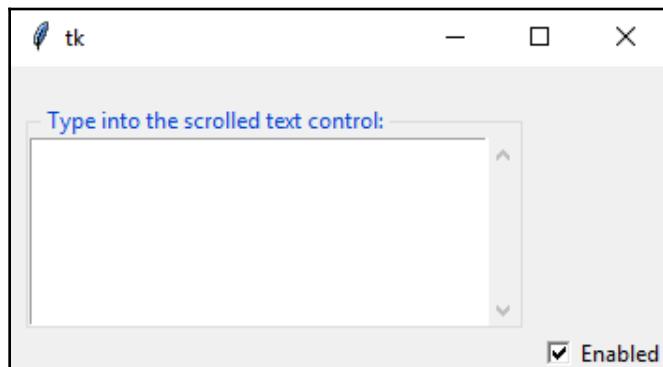
Having searched online and read the documentation, we might start by writing the following code to create our GUI:

1. Create a new module: `GUI_Spaghetti.py`.
2. Add the following code:

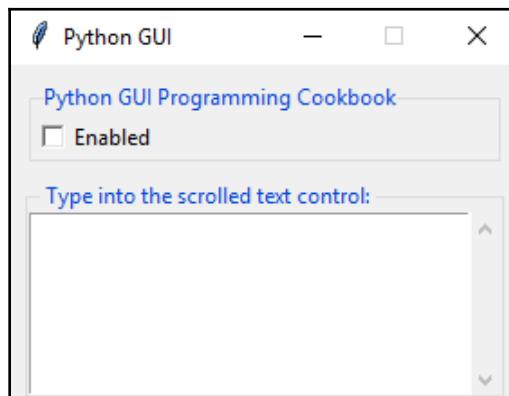
```
# Spaghetti Code #####  
def PRINTME(me):print(me)  
import tkinter
```

```
x=y=z=1
PRINTME(z)
from tkinter import *
scrolW=30;scrolH=6
win=tkinter.Tk()
if x:chVarUn=tkinter.IntVar()
from tkinter import ttk
WE='WE'
import tkinter.scrolledtext
outputFrame=tkinter.ttk.LabelFrame(win,text=' Type into the
scrolled text control: ')
scr=tkinter.scrolledtext.ScrolledText(outputFrame,width=scrolW,heig
ht=scrolH,wrap=tkinter.WORD)
e='E'
scr.grid(column=1,row=1,sticky=WE)
outputFrame.grid(column=0,row=2,sticky=e,padx=8)
lFrame=None
if
y:chck2=tkinter.Checkbutton(lFrame,text="Enabled",variable=chVarUn)
WE='WE'
if y==x:PRINTME(x)
lFrame=tkinter.ttk.LabelFrame(win,text="Spaghetti")
chck2.grid(column=1,row=4,sticky=tkinter.W,columnspan=3)
PRINTME(z)
lFrame.grid(column=0,row=0,sticky=wE,padx=10,pady=10)
chck2.select()
try: win.mainloop()
except:PRINTME(x)
chck2.deselect()
if y==x:PRINTME(x)
# End Pasta #####
```

- Run the code and observe the output, as follows:



4. Compare the preceding GUI to the intended GUI design, as follows:



5. Create a new module, `GUI_NOT_Spaghetti.py`, and add the following code:

```
#=====
# imports
#=====
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
#=====
# Create instance
#=====
win = tk.Tk()
#=====
# Add a title
#=====
win.title("Python GUI")
#=====
# Disable resizing the GUI
#=====
win.resizable(0,0)
```

6. Next, add some controls:

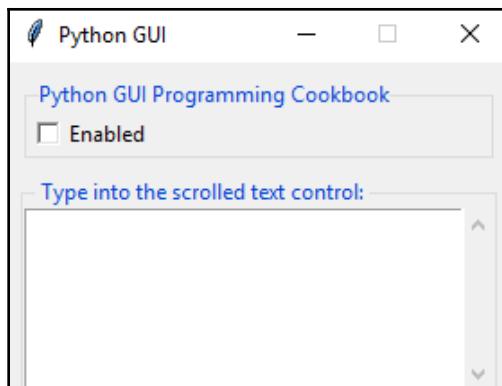
```
#=====
# Adding a LabelFrame, Textbox (Entry) and Combobox
#=====
lFrame = ttk.LabelFrame(win, text="Python GUI Programming
Cookbook")
lFrame.grid(column=0, row=0, sticky='WE', padx=10, pady=10)
#=====
# Using a scrolled Text control
```

```
#=====
outputFrame = ttk.LabelFrame(win, text=' Type into the scrolled
text
control: ')
outputFrame.grid(column=0, row=2, sticky='E', padx=8)
scrolW = 30
scrolH = 6
scr = scrolledtext.ScrolledText(outputFrame, width=scrolW,
height=scrolH, wrap=tk.WORD)
scr.grid(column=1, row=0, sticky='WE')
```

7. Add some more widgets:

```
#=====
# Creating a checkbutton
#=====
chVarUn = tk.IntVar()
check2 = tk.Checkbutton(lFrame, text="Enabled", variable=chVarUn)
check2.deselect()
check2.grid(column=1, row=4, sticky=tk.W, columnspan=3)
#=====
# Start GUI
#=====
win.mainloop()
```

8. Run the code and observe the following output:



Let's go behind the scenes to understand the code better.

How it works...

While the spaghetti code created a GUI, it is very hard to read because there is so much confusion in the code. Good code has many advantages over spaghetti code.

Let's see an example of spaghetti code first:

```
def PRINTME(me):print(me)
import tkinter
x=y=z=1
PRINTME(z)
from tkinter import *
```

Now, consider this example good code (note that there is not much confusion in reading the code):

```
#=====
# imports
#=====
import tkinter as tk
from tkinter import ttk
```

The good code has a clearly commented section. We can easily find the import statements:

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

Consider the following spaghetti code:

```
import tkinter.scrolledtext
outputFrame=tkinter.ttk.LabelFrame(win,text=' Type into the scrolled text
control: ')
scr=tkinter.scrolledtext.ScrolledText(outputFrame,width=scrolW,height=scrol
H,wrap=tkinter.WORD)
e='E'
scr.grid(column=1,row=1,sticky=WE)
outputFrame.grid(column=0,row=2,sticky=e,padx=8)
lFrame=None
if y:chck2=tkinter.Checkbutton(lFrame,text="Enabled",variable=chVarUn)
wE='WE'
if y==x:PRINTME(x)
lFrame=tkinter.ttk.LabelFrame(win,text="Spaghetti")
```

Now, consider the following good code. Here, as stated previously, we can easily find the import statements:

```
#=====
# Adding a LabelFrame, Textbox (Entry) and Combobox
#=====
```

```
lFrame = ttk.LabelFrame(win, text="Python GUI Programming Cookbook")
lFrame.grid(column=0, row=0, sticky='WE', padx=10, pady=10)

#=====
# Using a scrolled Text control
#=====
outputFrame = ttk.LabelFrame(win, text=' Type into the scrolled text
control: ')
outputFrame.grid(column=0, row=2, sticky='E', padx=8)
```

Good code, as shown in the preceding block, has a natural flow that follows how the widgets get laid out in the main GUI form.

In the spaghetti code, the bottom LabelFrame gets created before the top LabelFrame, and it is intermixed with an import statement and some widget creation:

```
#-----
```

The following is an example of spaghetti code that portrays this feature:

```
def PRINTME(me):print(me)
x=y=z=1
e='E'
WE='WE'
scr.grid(column=1, row=1, sticky=WE)
wE='WE'
if y==x:PRINTME(x)
lFrame.grid(column=0, row=0, sticky=wE, padx=10, pady=10)
PRINTME(z)
try: win.mainloop()
except:PRINTME(x)
chck2.deselect()
if y==x:PRINTME(x)
```

Good code does not contain unnecessary variable assignments, nor does it have a PRINTME function that does not do the debugging we might expect it to when reading the code:

```
#-----
```

The following code blocks enumerate this aspect.

Here is the spaghetti code:

```
import tkinter
x=y=z=1
PRINTME(z)
from tkinter import *
scrolW=30;scrolH=6
```

```
win=tkinter.Tk()
if x:chVarUn=tkinter.IntVar()
from tkinter import ttk
WE='WE'
import tkinter.scrolledtext
```

Here is the good code:

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
```

Good code has none of the instances mentioned for the spaghetti code.

The `import` statements only import the required modules, and they are not cluttered throughout the code. Also, there are no duplicate `import` statements. There is no `import *` statement:

```
#-----
```

The following code blocks enumerate this aspect.

This is the spaghetti code:

```
x=y=z=1
if x:chVarUn=tkinter.IntVar()
wE='WE'
```

Here is the good code:

```
=====
# Using a scrolled Text control
=====
outputFrame = ttk.LabelFrame(win, text=' Type into the scrolled text
control: ')
outputFrame.grid(column=0, row=2, sticky='E', padx=8)
scrolW = 30
scrolH = 6
scr = scrolledtext.ScrolledText(outputFrame, width=scrolW,
height=scrolH, wrap=tk.WORD)
scr.grid(column=1, row=0, sticky='WE')
```

Good code, as shown in the preceding example and compared to spaghetti code, has variable names that are quite meaningful. There are no unnecessary `if` statements that use the number 1 instead of True. It also has good indentation that makes the code much more readable.

In `GUI_NOT_Spaghetti.py`, we did not lose the intended window title and our check button ended up in the correct position. We also made the `LabelFrame` that surrounds the check button visible.

In `GUI_Spaghetti.py`, we both lost the window title and did not display the top `LabelFrame`. The check button ended up in the wrong place.

Using `__init__` to connect modules

When we create a new Python package using the PyDev plugin for the Eclipse IDE, it automatically creates an `__init__.py` module. We can also create it ourselves manually, when not using Eclipse.



The `__init__.py` module is usually empty and, then, has a size of 0 KB.

We can use this usually empty module to connect different Python modules by entering code into it. This recipe will show how to do this.

Getting ready

We will create a new GUI similar to the one we created in the previous recipe, *Avoiding spaghetti code*.

How to do it...

As our project becomes larger and larger, we naturally break it out into several Python modules. Sometimes, it can be complicated to find modules that are located in different subfolders, either above or below the code that needs to import it.

Let's see this recipe sequentially:

1. Create an empty file and save it as `__init__.py`.
2. Look at its size:

Ch11_Code	
Name	Size
<code>__init__.py</code>	0 KB
<code>GUI_NOT_Spaghetti.py</code>	2 KB
<code>GUI_Spaghetti.py</code>	2 KB

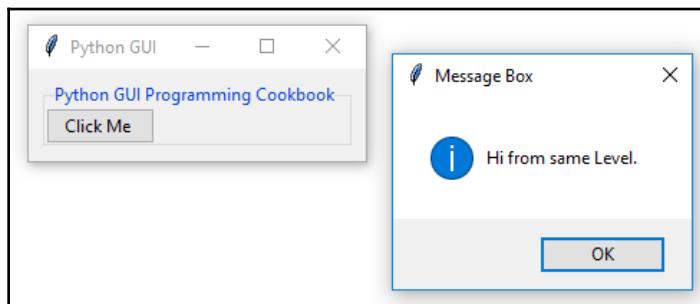
3. Create a new module, `GUI__init__.py`, and add the following code:

```
#=====
# imports
#=====
import tkinter as tk
from tkinter import ttk
#=====
# Create instance
#=====
win = tk.Tk()
#=====
# Add a title
#=====
win.title("Python GUI")
```

4. Next, add some widgets and a callback function:

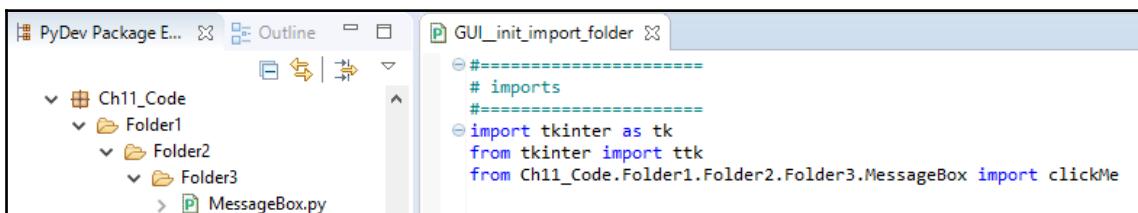
```
#=====
# Adding a LabelFrame and a Button
#=====
lFrame = ttk.LabelFrame(win, text="Python GUI Programming
Cookbook")
lFrame.grid(column=0, row=0, sticky='WE', padx=10, pady=10)
def clickMe():
    from tkinter import messagebox
    messagebox.showinfo('Message Box', 'Hi from same Level.')
button = ttk.Button(lFrame, text="Click Me ", command=clickMe)
button.grid(column=1, row=0, sticky=tk.S)
#=====
# Start GUI
#=====
win.mainloop()
```

5. Run the code and click the Click Me button:



6. Create three subfolders below where you are running your Python modules from.

7. Name them Folder1, Folder2, and Folder3:



8. In Folder3, create a new module: `MessageBox.py`.

9. Add the following code:

```
from tkinter import messagebox

def clickMe():
    messagebox.showinfo('Imported Message Box', 'Hi from Level 3')
```

10. Open `GUI_init.py` and save it as `GUI_init_import_folder.py`.

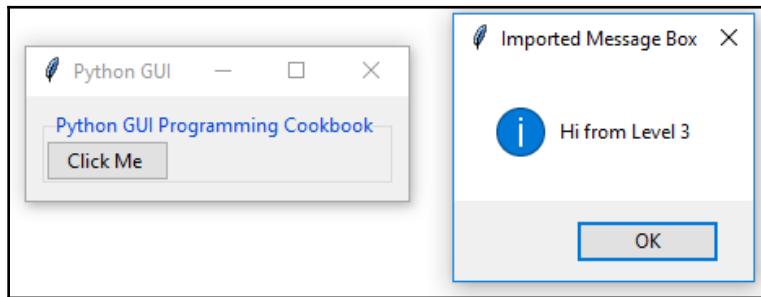
11. Add the following import:

```
from Ch11_Code.Folder1.Folder2.Folder3.MessageBox import clickMe
```

12. Comment out or delete the `clickMe` function:

```
# def clickMe():          # commented out
#     from tkinter import messagebox
#     messagebox.showinfo('Message Box', 'Hi from same Level.')
```

13. Run the code from your development environment and observe the output:



14. Open Command Prompt and try to run it. If running the code is unsuccessful, you can see the following output:

```
C:\WINDOWS\system32\cmd.exe

C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch11_Code>python GUI__init_import_folder.py
Traceback (most recent call last):
  File "GUI__init_import_folder.py", line 13, in <module>
    from Ch11_Code.Folder1.Folder2.Folder3.MessageBox import clickMe
ModuleNotFoundError: No module named 'Ch11_Code'

C:\Eclipse_NEON_workspace\2nd Edition Python GUI Programming Cookbook\Ch11_Code>
```

A screenshot of a Windows Command Prompt window. The command `python GUI__init_import_folder.py` is run, resulting in a `ModuleNotFoundError` because the module 'Ch11_Code' is not found.

15. Open `__init__.py`.

16. Add the following code to the `__init__.py` module:

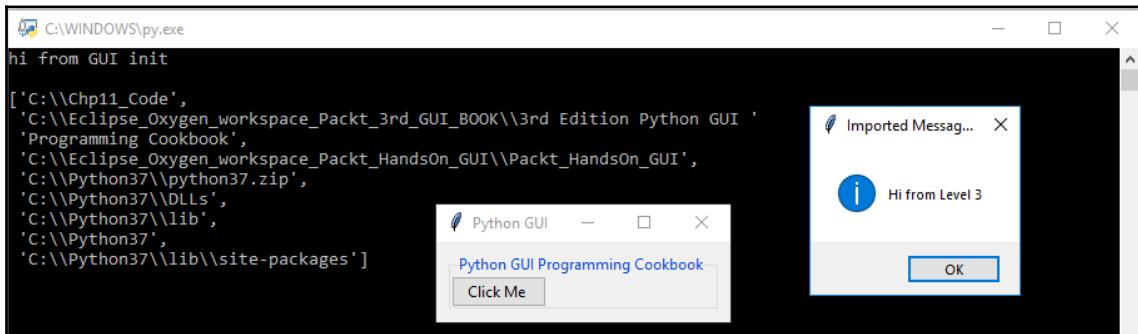
```
print('hi from GUI init\n')
from sys import path
from pprint import pprint
=====
#
# Required setup for the PYTHONPATH in order to find all package
# folders
#
#
from site import addsitedir
from os import.getcwd, chdir, pardir
while True:
    curFull = getcwd()
    curDir = curFull.split('\\')[-1]
    if 'Ch11_Code' == curDir:
        addsitedir(curFull)
        addsitedir(curFull + 'Folder1\Folder2\Folder3')
        break
```

```
    chdir(pardir)
    pprint(path)
```

17. Open `GUI__init__import_folder.py` and save it as `GUI__init__import_folder_directly.py`.
18. Add the following two import statements and comment out the previous import:

```
# from Ch11_Code.Folder1.Folder2.Folder3.MessageBox import clickMe
# comment out
import __init__
from MessageBox import clickMe
```

19. Run the code from Command Prompt:



Let's go behind the scenes to understand the code better.

How it works...

When we create an `__init__.py` module, it is typically empty with a file size of 0 KB.



The `__init__.py` module is not the same as the `__init__(self):` method of a Python class.

In `GUI__init__.py`, we created the following function, which imports Python's message box and then uses it to display the message box dialog window:

```
def clickMe():
    from tkinter import messagebox
    messagebox.showinfo('Message Box', 'Hi from same Level.')
```

When we move the `clickMe()` message box code into a nested directory folder and try to import it into our GUI module, we might run into some challenges.

We have created three subfolders below where our Python module lives. We have then placed the `clickMe()` message box code into a new Python module, which we named `MessageBox.py`. This module is located in `Folder3`, three levels below where our Python module is.

We want to import `MessageBox.py` in order to use the `clickMe()` function that this module contains.

We can use Python's relative import syntax:

```
from Ch11_Code.Folder1.Folder2.Folder3.MessageBox import clickMe
```

In the preceding code, the path is hardcoded. If we remove `Folder2`, it would no longer work.

In `GUI__init__import_folder.py`, we deleted the local `clickMe()` function and now our callback is expected to use the imported `clickMe()` function. This works from within Eclipse and other IDEs that set `PYTHONPATH` to a project where you develop your code.

It may or may not work from Command Prompt, depending on whether you have set `PYTHONPATH` to the root of where the `Ch11_Code\Folder1\Folder2\Folder3` folders are located.

To solve this error, we can initialize our Python search path from within the `__init__.py` module. This often solves relative import errors.

In the `GUI__init__import_folder_directly.py` module, we no longer have to specify the full folder path. We can import the module and its function directly.



We have to explicitly import `__init__` for this code to work.

This recipe showed a few troubleshooting approaches in case you run into this sort of challenge.

Mixing fall-down and OOP coding

Python is an OOP language, yet it does not always make sense to use OOP. For simple scripting tasks, the legacy waterfall coding style is still appropriate.

In this recipe, we will create a new GUI that mixes both the fall-down coding style with the more modern OOP coding style.

We will create an OOP-style class that will display a tooltip when we hover the mouse over a widget in a Python GUI, which we will create using the waterfall style.



Fall-down and waterfall coding styles are the same. It means that we have to physically place code above code before we can call it from the code below. In this paradigm, the code literally falls down from the top of our program to the bottom of our program when we execute the code.

Getting ready

In this recipe, we will create a GUI using `tkinter`, which is similar to the GUI we created in the first chapter of this book.

How to do it...

Let's see how to perform this recipe:

1. We will first create a GUI using `tkinter` in a procedural fashion and then we will add a class to it to display tooltips over GUI widgets.
2. Next, we will create a new module: `GUI_FallDown.py`.
3. Add the following code:

```
#=====
# imports
#=====
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
#=====
# Create instance
#=====
win = tk.Tk()
#=====
# Add a title
```

```
#=====
win.title("Python GUI")
#=====
# Disable resizing the GUI
#=====
win.resizable(0,0)
```

4. Next, add some widgets:

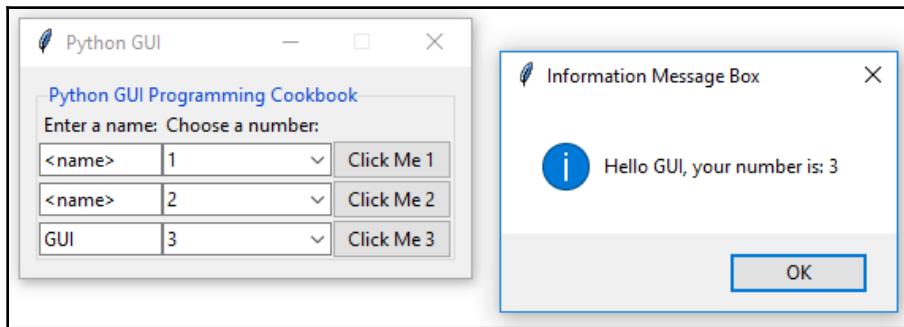
```
#=====
# Adding a LabelFrame, Textbox (Entry) and Combobox
#=====
lFrame = ttk.LabelFrame(win, text="Python GUI Programming
Cookbook")
lFrame.grid(column=0, row=0, sticky='WE', padx=10, pady=10)
#=====
# Labels
#=====
ttk.Label(lFrame, text="Enter a name:").grid(column=0, row=0)
ttk.Label(lFrame, text="Choose a number:").grid(column=1, row=0,
sticky=tk.W)
#=====
# Buttons click command
#=====
def clickMe(name, number):
    messagebox.showinfo('Information Message Box', 'Hello '+name+
    ', your number is: ' + number)
```

5. Add more widgets in a loop:

```
#=====
# Creating several controls in a loop
#=====
names      = ['name0', 'name1', 'name2']
nameEntries = ['nameEntry0', 'nameEntry1', 'nameEntry2']
numbers    = ['number0', 'number1', 'number2']
numberEntries = ['numberEntry0', 'numberEntry1', 'numberEntry2']
buttons = []
for idx in range(3):
    names[idx] = tk.StringVar()
    nameEntries[idx] = ttk.Entry(lFrame, width=12,
textvariable=names[idx])
    nameEntries[idx].grid(column=0, row=idx+1)
    nameEntries[idx].delete(0, tk.END)
    nameEntries[idx].insert(0, '<name>')
    numbers[idx] = tk.StringVar()
    numberEntries[idx] = ttk.Combobox(lFrame, width=14,
textvariable=numbers[idx])
```

```
numberEntries[idx]['values'] = (1+idx, 2+idx, 4+idx, 42+idx,
100+idx)
numberEntries[idx].grid(column=1, row=idx+1)
numberEntries[idx].current(0)
button = ttk.Button(lFrame, text="Click Me "+str(idx+1),
command=lambda idx=idx: clickMe(names[idx].get(),
numbers[idx].get()))
button.grid(column=2, row=idx+1, sticky=tk.W)
buttons.append(button)
=====
# Start GUI
=====
win.mainloop()
```

6. Run the code and click one of the buttons:

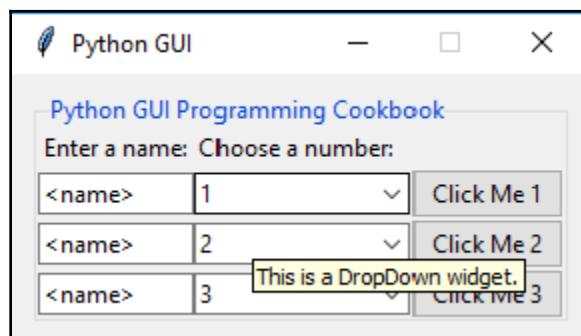


7. Create a new module: `GUI_FallDown_Tooltip.py`.
8. Use the code from `GUI_FallDown.py` and then add the following code to it at the top:

```
import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
=====
#
# Add this code at the top
class ToolTip(object):
    def __init__(self, widget, tip_text=None):
        self.widget = widget
        self.tip_text = tip_text
        widget.bind('<Enter>', self.mouse_enter)
        widget.bind('<Leave>', self.mouse_leave)
    def mouse_enter(self, _event):
        self.show_tooltip()
```

```
def mouse_leave(self, _event):
    self.hide_tooltip()
def show_tooltip(self):
    if self.tip_text:
        x_left = self.widget.winfo_rootx()
        # get widget top-left coordinates
        y_top = self.widget.winfo_rooty() - 18
        # place tooltip above widget
        self.tip_window = tk.Toplevel(self.widget)
        self.tip_window.overrideredirect(True)
        self.tip_window.geometry("%d%d" % (x_left, y_top))
        label = tk.Label(self.tip_window, text=self.tip_text,
                          justify=tk.LEFT, background="#ffffe0",
                          relief=tk.SOLID, borderwidth=1,
                          font=("tahoma", "8", "normal"))
        label.pack(ipadx=1)
    def hide_tooltip(self):
        if self.tip_window:
            self.tip_window.destroy()
#=====
# ...
# Add this code at the bottom
# Add Tooltips to widgets
ToolTip(nameEntries[0], 'This is an Entry widget.')
ToolTip(numberEntries[0], 'This is a DropDown widget.')
ToolTip(buttons[0], 'This is a Button widget.')
#=====
# Start GUI
#=====
win.mainloop()
```

9. Run the code and hover the mouse over several widgets:



Let's go behind the scenes to understand the code better.

How it works...

First, we create a Python GUI in `GUI_FallDown.py` using `tkinter` and code it in the waterfall style.

We can improve our Python GUI by adding tooltips. The best way to do this is to isolate the code that creates the tooltip functionality from our GUI.

We do this by creating a separate class, which has the tooltip functionality, and then we create an instance of this class in the same Python module that creates our GUI.

Using Python, there is no need to place our `ToolTip` class into a separate module. We can place it just above the procedural code and then call it from below the class code.

In `GUI_FallDown_Tooltip.py`, the code is almost identical to `GUI_FallDown.py`, but now we have tooltips.

We can very easily mix and match both procedural and OOP programming in the same Python module.

Using a code naming convention

This recipe will show you the value of adhering to a code naming scheme: it helps us to find the code we want to extend, and reminds us of the design of our program.

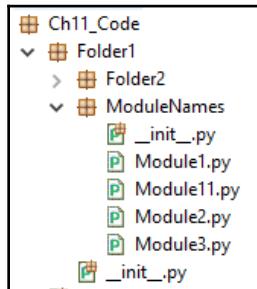
Getting ready

In this recipe, we will look at Python module names and look at good naming conventions.

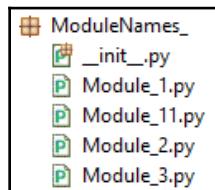
How to do it...

We will create example projects with different Python module names to compare the naming:

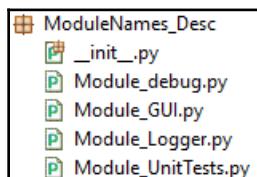
1. Create a new `ModuleNames` folder under `Folder1`.
2. Add the following Python modules, 1, 11, 2, and 3:



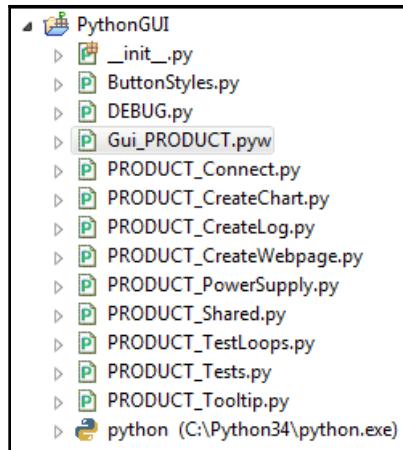
3. Next, create a new `ModuleNames_` folder under `Folder1`.
4. Add the following Python modules, 1, 11, 2, and 3:



5. Next, create a new `ModuleNames_Desc` folder under `Folder1`.
6. Add the following Python modules, `Logger`, `UnitTests`, `GUI`, and `debug`:



7. Look at this naming convention as an example:



Let's go behind the scenes to understand the code better.

How it works...

In *step 1*, we create a package subfolder named `ModuleNames`.

In *step 2*, we add Python modules to it.

In *step 3*, we create another package folder and add a trailing underscore to the name: `ModuleNames_`.

In *step 4*, we add new Python modules that have the same names as the ones in *step 2*.

In *step 5*, we create another package folder with a much more descriptive name, `ModuleNames_Desc`.

In *step 6*, we add Python modules but this time with much more descriptive names that explain the purpose of each Python module.

Lastly, in *step 7*, we show a full example of how this can look.

There's more...

Often, a typical way to start coding is by incrementing numbers, as can be seen in `ModuleNames`.

Later, coming back to this code, we don't have much of an idea which Python module provides which functionality and, sometimes, our last incremented modules are not as good as the earlier versions.

A clear naming convention does help.



A slight improvement is adding underscores, which makes module names more readable, as in `ModuleNames_`.

A better way is to add some description of what the module does, as seen in `ModuleNames_Desc`.

While not perfect, the names chosen for the different Python modules indicate what each module's responsibility is. When we want to add more unit tests, it is clear in which module they reside.

In the last example, we are using the placeholder, `PRODUCT`, for a real name.

Replace the word `PRODUCT` with the product you are currently working on.



The entire application is a GUI. All parts are connected. The `DEBUG.py` module is only used for debugging code. The main module to invoke the GUI has its name reversed when compared with all of the other modules. It starts with `Gui` and has a `.pyw` extension.

It is the only Python module that has this extension name.

From this naming convention, if you are familiar enough with Python, it will be obvious that, to run this GUI, you can double-click the `Gui_PRODUCT.pyw` module.

All other Python modules contain functionality for the GUI and also execute the underlying business logic to fulfill the purpose this GUI addresses.

Naming conventions for Python code modules are a great help in keeping us efficient and helping us to remember our original design. When we need to debug and fix a defect or add new functionality, they are the first resources to look at.



Incrementing module names by numbers is not very meaningful and eventually wastes development time.

On the other hand, naming Python variables is more of a free form. Python infers types, so we do not have to specify that a variable will be of the `list` type (it might not be, or later in the code, it might become a different type).

A good idea for naming variables is to make them descriptive, and it is a good idea not to abbreviate too much.

If we wish to point out that a certain variable is designed to be of the `list` type, then it is much more intuitive to use the full word `list_of_things` instead of `lst_of_things`. Similarly, use for `numbers` instead of `num`.

While it is a good idea to have very descriptive names for variables, sometimes that can get too long. In Apple's Objective-C language, some variable and function names are extreme: `thisIsAMethodThatDoesThisAndThatAndAlsoThatIfYouPassInNIntegers:1:2:3`.



Use common sense when naming variables, methods, and functions.

Now, let's move on to the next recipe.

When not to use OOP

Python comes built in with OOP capabilities, but at the same time, we can write scripts that do not need to use OOP. For some tasks, OOP does not make sense.

This recipe will show us when not to use OOP.

Getting ready

In this recipe, we will create a Python GUI similar to the previous recipes. We will compare the OOP code to the non-OOP alternative way of programming the GUI. The resultant output will be the same but the code of the two versions is slightly different.

How to do it...

Let's see how to perform this recipe:

1. Let's first create a new GUI using the OOP methodology. The code shown in the following steps will create the GUI displayed.
2. Create a new module: `GUI_OOP.py`.
3. Add the following code:

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
```

4. Create a class:

```
class OOP():
    def __init__(self):
        self.win = tk.Tk()
        self.win.title("Python GUI")
        self.createWidgets()
```

5. Add a method to create widgets:

```
def createWidgets(self):
    tabControl = ttk.Notebook(self.win)
    tab1 = ttk.Frame(tabControl)
    tabControl.add(tab1, text='Tab 1')
    tabControl.pack(expand=1, fill="both")
    self.monty = ttk.LabelFrame(tab1, text=' Mighty Python ')
    self.monty.grid(column=0, row=0, padx=8, pady=4)

    ttk.Label(self.monty, text="Enter a name:").grid(column=0,
    row=0, sticky='W')
    self.name = tk.StringVar()
    nameEntered = ttk.Entry(self.monty, width=12,
    textvariable=self.name)
    nameEntered.grid(column=0, row=1, sticky='W')

    self.action = ttk.Button(self.monty, text="Click Me!")
    self.action.grid(column=2, row=1)

    ttk.Label(self.monty, text="Choose a number:")
    .grid(column=1, row=0)
    number = tk.StringVar()
    numberChosen = ttk.Combobox(self.monty, width=12,
    textvariable=number)
```

```
numberChosen['values'] = (42)
numberChosen.grid(column=1, row=1)
numberChosen.current(0)

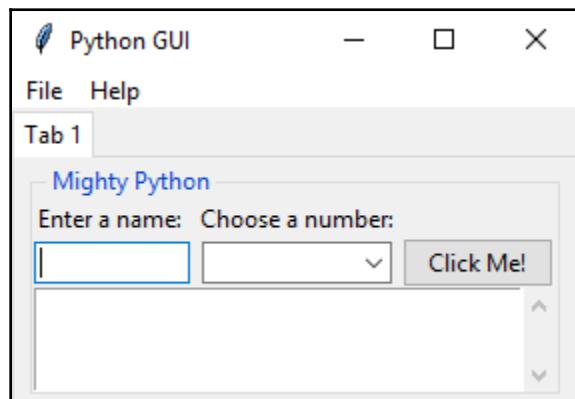
scrolW = 30; scrolH = 3
self.scr = scrolledtext.ScrolledText(self.monty, width=scrolW,
height=scrolH, wrap=tk.WORD)
self.scr.grid(column=0, row=3, sticky='WE', columnspan=3)
```

6. Create a menu bar:

```
menuBar = Menu(tab1)
self.win.config(menu=menuBar)
fileMenu = Menu(menuBar, tearoff=0)
menuBar.add_cascade(label="File", menu=fileMenu)
helpMenu = Menu(menuBar, tearoff=0)
menuBar.add_cascade(label="Help", menu=helpMenu)

nameEntered.focus()
=====
oop = OOP()
oop.win.mainloop()
```

7. Run the code and observe the following output:



Let's have a look at a new scenario:

1. Create a new module, `GUI_NOT_OOP.py`, and add the following code:

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
```

```
def createWidgets():
    tabControl = ttk.Notebook(win)
    tab1 = ttk.Frame(tabControl)
    tabControl.add(tab1, text='Tab 1')
    tabControl.pack(expand=1, fill="both")
    monty = ttk.LabelFrame(tab1, text=' Mighty Python ')
    monty.grid(column=0, row=0, padx=8, pady=4)
```

2. Create more widgets:

```
ttk.Label(monty, text="Enter a name:").grid(column=0, row=0,
                                             sticky='W')
name = tk.StringVar()
nameEntered = ttk.Entry(monty, width=12, textvariable=name)
nameEntered.grid(column=0, row=1, sticky='W')

action = ttk.Button(monty, text="Click Me!")
action.grid(column=2, row=1)

ttk.Label(monty, text="Choose a number:").grid(column=1, row=0)
number = tk.StringVar()
numberChosen = ttk.Combobox(monty, width=12, textvariable=number)
numberChosen['values'] = (42)
numberChosen.grid(column=1, row=1)
numberChosen.current(0)

scrolW = 30; scrolH = 3
scr = scrolledtext.ScrolledText(monty, width=scrolW,
                               height=scrolH, wrap=tk.WORD)
scr.grid(column=0, row=3, sticky='WE', columnspan=3)
```

3. Create a menu bar:

```
menuBar = Menu(tab1)
win.config(menu=menuBar)
fileMenu = Menu(menuBar, tearoff=0)
menuBar.add_cascade(label="File", menu=fileMenu)
helpMenu = Menu(menuBar, tearoff=0)
menuBar.add_cascade(label="Help", menu=helpMenu)

nameEntered.focus()
```

4. Now, create the entire GUI, calling the function that creates the widgets:

```
#=====
win = tk.Tk()
win.title("Python GUI")
createWidgets()
win.mainloop()
```

5. Run the code. The resultant GUI will be identical to the one from `GUI_OOP.py` shown previously.

How it works...

First, we create a Python `tkinter` GUI in OOP style, `GUI_OOP.py`. Then, we create the same GUI in a procedural style, `GUI_NOT_OOP.py`.

We can achieve the same GUI without using an OOP approach by restructuring our code slightly. First, we remove the `OOP` class and its `__init__` method. Next, we move all of the methods to the left and remove the `self` class reference, which turns them into unbound functions. We also remove any other `self` references our previous code had. Then, we move the `createWidgets` function call below the point of the function's declaration. We place it just above the `mainloop` call.

In the end, we achieve the same GUI but without using OOP.

Python enables us to use OOP when it makes sense. Other languages such as Java and C# force us to always use the OOP approach to coding. In this recipe, we explored a situation when it did not make sense to use OOP.



The OOP approach will be more extendable if the code base grows, but if it's certain that it is the only code that's needed, then there's no need to go through OOP.

Now, let's move on to the next recipe.

How to use design patterns successfully

In this recipe, we will create widgets for our Python GUI by using the *factory design pattern*. In the previous recipes, we created our widgets either manually one at a time or dynamically in a loop. Using the factory design pattern, we will use the *factory* to create our widgets.

Getting ready

We will create a Python GUI that has three buttons, each having a different style.

How to do it...

We will create a Python GUI with different button styles and we will use a factory design pattern to create these different styles:

1. Create a new module: `GUI_DesignPattern.py`.
2. Add the following code:

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
```

3. Create the factory class:

```
class ButtonFactory():
    def createButton(self, type_):
        return buttonTypes[type_]()
```

4. Create a base class:

```
class ButtonBase():
    relief = 'flat'
    foreground = 'white'
    def getButtonConfig(self):
        return self.relief, self.foreground
```

5. Create classes that inherit from the base class:

```
class ButtonRidge(ButtonBase):
    relief = 'ridge'
    foreground = 'red'
```

```
class ButtonSunken(ButtonBase):
    relief ='sunken'
    foreground ='blue'

class ButtonGroove(ButtonBase):
    relief ='groove'
    foreground ='green'
```

6. Create a list that contains the previous classes:

```
buttonTypes = [ButtonRidge, ButtonSunken, ButtonGroove]
```

7. Create a new class that uses the previous code:

```
class OOP():
    def __init__(self):
        self.win = tk.Tk()
        self.win.title("Python GUI")
        self.createWidgets()

    def createWidgets(self):
        tabControl = ttk.Notebook(self.win)
        tab1 = ttk.Frame(tabControl)
        tabControl.add(tab1, text='Tab 1')
        tabControl.pack(expand=1, fill="both")
        self.monty = ttk.LabelFrame(tab1, text=' Monty Python ')
        self.monty.grid(column=0, row=0, padx=8, pady=4)
        scr = scrolledtext.ScrolledText(self.monty, width=30,
                                        height=3, wrap=tk.WORD)
        scr.grid(column=0, row=3, sticky='WE', columnspan=3)
        menuBar = Menu(tab1)
        self.win.config(menu=menuBar)
        fileMenu = Menu(menuBar, tearoff=0)
        menuBar.add_cascade(label="File", menu=fileMenu)
        helpMenu = Menu(menuBar, tearoff=0)
        menuBar.add_cascade(label="Help", menu=helpMenu)
        self.createButtons()

    def createButtons(self):
        factory = ButtonFactory() # <-- create the factory
        # Button 1
        rel = factory.createButton(0).getButtonConfig()[0]
        fg = factory.createButton(0).getButtonConfig()[1]
        action = tk.Button(self.monty, text="Button "+str(0+1),
                           relief=rel, foreground=fg)
        action.grid(column=0, row=1)
        # Button 2
        rel = factory.createButton(1).getButtonConfig()[0]
```

```
fg = factory.createButton(1).getButtonConfig() [1]
action = tk.Button(self.monty, text="Button "+str(1+1),
relief=rel, foreground=fg)
action.grid(column=1, row=1)
# Button 3
rel = factory.createButton(2).getButtonConfig() [0]
fg = factory.createButton(2).getButtonConfig() [1]
action = tk.Button(self.monty, text="Button "+str(2+1),
relief=rel, foreground=fg)
action.grid(column=2, row=1)
=====
oop = OOP()
oop.win.mainloop()
```

8. Run the code and observe the output:



Let's go behind the scenes to understand the code better.

How it works...

We create a base class that our different button style classes inherit from and in which each of them overrides the `relief` and `foreground` configuration attributes. All subclasses inherit the `getButtonConfig` method from this base class. This method returns a tuple.

We also create a button factory class and a list that holds the names of our button subclasses. We name the list `buttonTypes`, as our factory will create different types of buttons.

Further down in the module, we create the button widgets, using the same `buttonTypes` list. We create an instance of the button factory and then we use our factory to create our buttons.



The items in the `buttonTypes` list are the names of our subclasses.

We invoke the `createButton` method and then immediately call the `getButtonConfig` method of the base class and retrieve the configuration attributes using dot notation.

We can see that our Python GUI factory did indeed create different buttons, each having a different style. They differ in the color of their text and their `relief` property.

Design patterns are a very exciting tool in our software development toolbox.

Avoiding complexity

In this recipe, we will extend our Python GUI and learn ways to handle the increasing complexity of our software development efforts.

Our co-workers and clients love the GUIs we create in Python and ask for more and more features to add to our GUI.

This increases complexity and can easily ruin our original nice design.

Getting ready

We will create a new Python GUI similar to those in the previous recipes and will add many features to it in the form of widgets.

How to do it...

Let's see how to perform the recipe:

1. We will start with a Python GUI that has two tabs and then we will add more widgets to it.
2. Create a new module: `GUI_Complexity_start.py`.
3. Add the following code:

```
#=====
# imports
#=====
```

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
from tkinter import Spinbox
from Ch11_Code.ToolTip import ToolTip
```

4. Create a global variable and a class:

```
GLOBAL_CONST = 42
=====
=
class OOP():
    def __init__(self):
        # Create instance
        self.win = tk.Tk()
        # Add a title
        self.win.title("Python GUI")
        self.createWidgets()

    # Button callback
    def clickMe(self):
        self.action.configure(text='Hello ' + self.name.get())
    # Button callback Clear Text
    def clearScrol(self):
        self.scr.delete('1.0', tk.END)
    # Spinbox callback
    def _spin(self):
        value = self.spin.get()
        print(value)
        self.scr.insert(tk.INSERT, value + '\n')
    # Checkbox callback
    def checkCallback(self, *ignoredArgs):
        # only enable one checkbutton
        if self.chVarUn.get():
            self.check3.configure(state='disabled')
        else: self.check3.configure(state='normal')
        if self.chVarEn.get():
            self.check2.configure(state='disabled')
        else: self.check2.configure(state='normal')
    # Radiobutton callback function
    def radCall(self):
        radSel=self.radVar.get()
        if radSel == 0: self.monty2.configure(text='Blue')
        elif radSel == 1: self.monty2.configure(text='Gold')
        elif radSel == 2: self.monty2.configure(text='Red')
    # Exit GUI cleanly
    def _quit(self):
        self.win.quit()
```

```
        self.win.destroy()
        exit()
def usingGlobal(self):
    GLOBAL_CONST = 777
    print(GLOBAL_CONST)
```

5. Add a method that creates the widgets:

```
#####
#####
def createWidgets(self):
    tabControl = ttk.Notebook(self.win) # Create Tab Control
    tab1 = ttk.Frame(tabControl) # Create a tab
    tabControl.add(tab1, text='Tab 1') # Add the tab
    tab2 = ttk.Frame(tabControl) # Add a second tab
    tabControl.add(tab2, text='Tab 2') # Make second tab visible
    tabControl.pack(expand=1, fill="both") # Pack to make visible
    self.monty = ttk.LabelFrame(tab1, text=' Mighty Python ')
    self.monty.grid(column=0, row=0, padx=8, pady=4)

    ttk.Label(self.monty, text="Enter a name:").grid(column=0,
    row=0, sticky='W')
    self.name = tk.StringVar()
    nameEntered = ttk.Entry(self.monty, width=12,
    textvariable=self.name)
    nameEntered.grid(column=0, row=1, sticky='W')

    self.action = ttk.Button(self.monty, text="Click Me!",
    command=self.clickMe)
    self.action.grid(column=2, row=1)
    ttk.Label(self.monty, text="Choose a number:").grid(column=1,
    row=0)
    number = tk.StringVar()
    numberChosen = ttk.Combobox(self.monty, width=12,
    textvariable=number)
    numberChosen['values'] = (1, 2, 4, 42, 100)
    numberChosen.grid(column=1, row=1)
    numberChosen.current(0)

    self.spin = Spinbox(self.monty, values=(1, 2, 4, 42, 100),
    width=5, bd=8, command=self._spin)
    self.spin.grid(column=0, row=2)

    scrolW = 30; scrolH = 3
    self.scr = scrolledtext.ScrolledText(self.monty, width=scrolW,
    height=scrolH, wrap=tk.WORD)
    self.scr.grid(column=0, row=3, sticky='WE', columnspan=3)
```

```
    self.monty2 = ttk.LabelFrame(tab2, text=' Holy Grail ')
    self.monty2.grid(column=0, row=0, padx=8, pady=4)

    chVarDis = tk.IntVar()
    check1 = tk.Checkbutton(self.monty2, text="Disabled",
                           variable=chVarDis, state='disabled')
    check1.select()
    check1.grid(column=0, row=0, sticky=tk.W)
    self.chVarUn = tk.IntVar()
    self.check2 = tk.Checkbutton(self.monty2, text="UnChecked",
                                variable=self.chVarUn)
    self.check2.deselect()
    self.check2.grid(column=1, row=0, sticky=tk.W)
    self.chVarEn = tk.IntVar()
    self.check3 = tk.Checkbutton(self.monty2, text="Toggle",
                                variable=self.chVarEn)
    self.check3.deselect()
    self.check3.grid(column=2, row=0, sticky=tk.W)

    self.chVarUn.trace('w', lambda unused0, unused1, unused2 :
self.checkCallback())
    self.chVarEn.trace('w', lambda unused0, unused1, unused2 :
self.checkCallback())

colors = ["Blue", "Gold", "Red"]
self.radVar = tk.IntVar()
self.radVar.set(99)

for col in range(3):
    curRad = 'rad' + str(col)
    curRad = tk.Radiobutton(self.monty2, text=colors[col],
                           variable=self.radVar, value=col, command=self.radCall)
    curRad.grid(column=col, row=6, sticky=tk.W, columnspan=3)
    ToolTip(curRad, 'This is a Radiobutton control.')
labelsFrame = ttk.LabelFrame(self.monty2,
text=' Labels in a Frame ')
labelsFrame.grid(column=0, row=7)

ttk.Label(labelsFrame, text="Label1").grid(column=0, row=0)
ttk.Label(labelsFrame, text="Label2").grid(column=0, row=1)

for child in labelsFrame.winfo_children():
    child.grid_configure(padx=8)
menuBar = Menu(tab1)
self.win.config(menu=menuBar)

fileMenu = Menu(menuBar, tearoff=0)
fileMenu.add_command(label="New")
```

```
fileMenu.add_separator()
fileMenu.add_command(label="Exit", command=self._quit)
menuBar.add_cascade(label="File", menu=fileMenu)

helpMenu = Menu(menuBar, tearoff=0)
helpMenu.add_command(label="About")
menuBar.add_cascade(label="Help", menu=helpMenu)

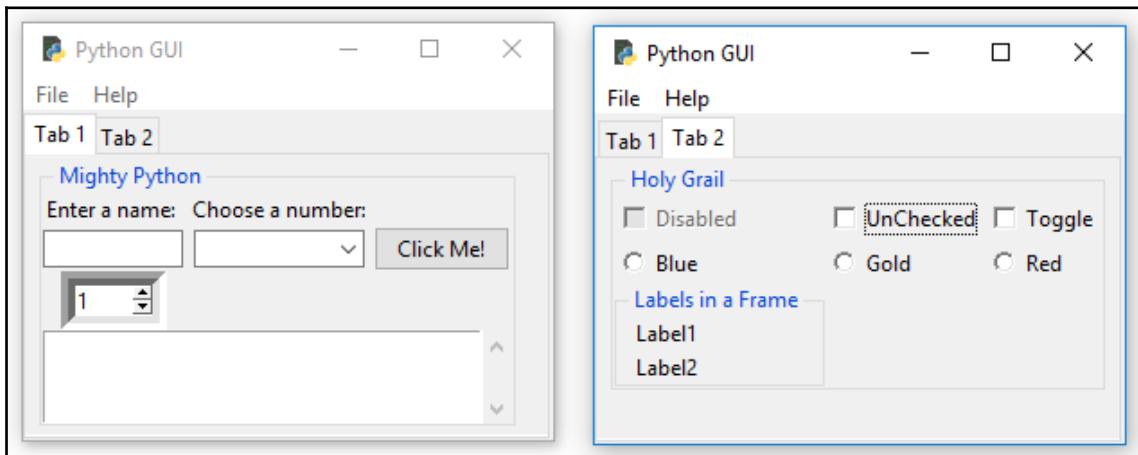
self.win.iconbitmap('pyc.ico')

strData = tk.StringVar()
strData.set('Hello StringVar')
intData = tk.IntVar()
strData = tk.StringVar()
strData = self.spin.get()
self.usingGlobal()
nameEntered.focus()

ToolTip(self.spin, 'This is a Spin control.')
ToolTip(nameEntered, 'This is an Entry control.')
ToolTip(self.action, 'This is a Button control.')
ToolTip(self.scr, 'This is a ScrolledText control.')
#=====
# Start GUI
#=====

oop = OOP()
oop.win.mainloop()
```

6. Run the code and click both tabs:



7. Open `GUI_Complexity_start.py` and save it as `GUI_Complexity_start_add_button.py`.
8. Add the following code to the `createWidgets` method:

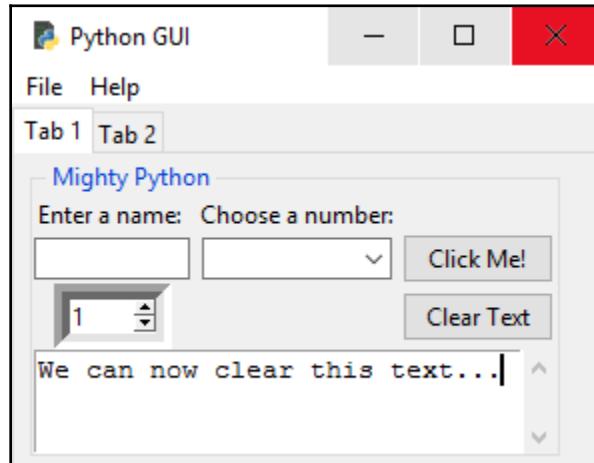
```
# Adding another Button
self.action = ttk.Button(self.monty, text="Clear Text",
command=self.clearScrol)
self.action.grid(column=2, row=2)
```

9. Add the following code just below `__init__(self)`:

```
# Button callback
def clickMe(self):
    self.action.configure(text='Hello ' + self.name.get())

# Button callback Clear Text
def clearScrol(self):
    self.scr.delete('1.0', tk.END)
```

10. Run the code and observe the following output:

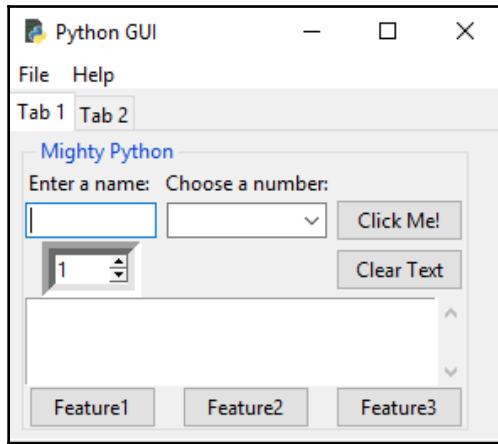


11. Open `GUI_Complexity_start_add_button.py` and save it as `GUI_Complexity_start_add_three_more_buttons.py`.

12. Add the following code to the `createWidgets` method:

```
# Adding more Feature Buttons
for idx in range(3):
    b = ttk.Button(self.monty, text="Feature" + str(idx+1))
    b.grid(column=idx, row=4)
```

13. Run the code and observe the output:



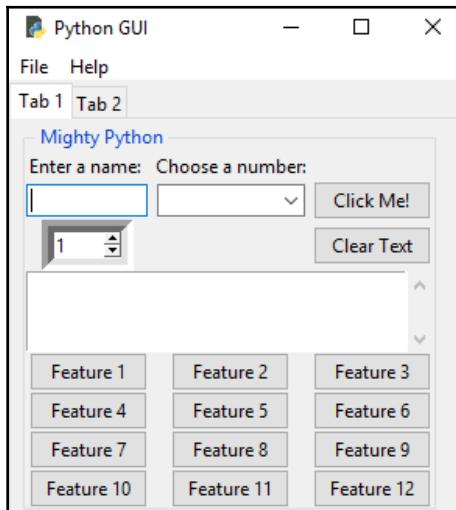
14. Open `GUI_Complexity_start_add_three_more_buttons.py` and save it as `GUI_Complexity_start_add_three_more_buttons_add_more.py`.

15. Add the following code to the `createWidgets` method:

```
# Adding more Feature Buttons
startRow = 4
for idx in range(12):
    if idx < 2: col = idx
    else: col += 1
    if not idx % 3:
        startRow += 1
        col = 0

    b = ttk.Button(self.monty, text="Feature " + str(idx+1))
    b.grid(column=col, row=startRow)
```

16. Run the code and observe the following output:



17. Open `GUI_Complexity_start_add_three_more_buttons_add_more.py` and save it as `GUI_Complexity_end_tab3.py`.
18. Add the following code to the `createWidgets` method:

```
# Tab Control 3 -----
tab3 = ttk.Frame(tabControl) # Add a tab
tabControl.add(tab3, text='Tab 3') # Make tab visible

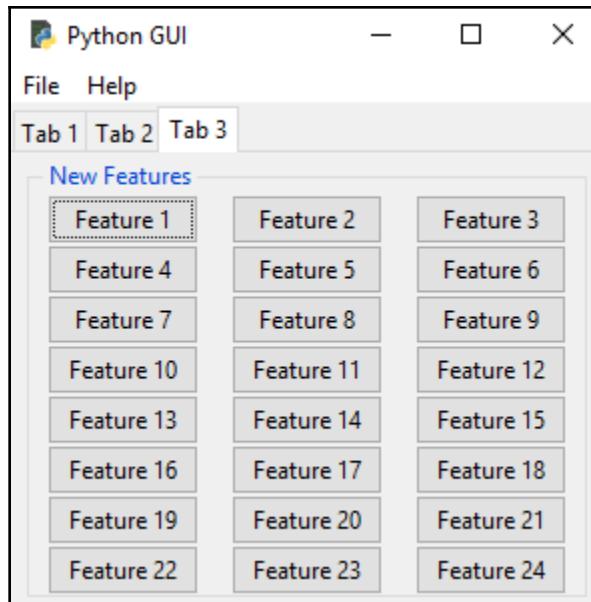
monty3 = ttk.LabelFrame(tab3, text=' New Features ')
monty3.grid(column=0, row=0, padx=8, pady=4)

# Adding more Feature Buttons
startRow = 4
for idx in range(24):
    if idx < 2: col = idx
    else: col += 1
    if not idx % 3:
        startRow += 1
        col = 0

    b = ttk.Button(monty3, text="Feature " + str(idx+1))
    b.grid(column=col, row=startRow)

# Add some space around each label
for child in monty3.winfo_children():
    child.grid_configure(padx=8)
```

19. Run the code and click on **Tab 3**:



Let's go behind the scenes to understand the code better.

How it works...

We start with a GUI built with `tkinter`, `GUI_Complexity_start.py`, and it has some widgets on two tabs. We have created similar GUIs throughout this entire book.

The first new feature request we receive is to add functionality to **Tab 1**, which clears the `scrolledtext` widget.

Easy enough. We just add another button to **Tab 1**.

We also have to create the callback method in `GUI_Complexity_start_add_button.py` to add the desired functionality, which we define toward the top of our class and outside the method that creates our widgets. Now, our GUI has a new button and, when we click it, we clear the text of the `ScrolledText` widget. To add this functionality, we had to add code in two places in the same Python module.

We inserted the new button in the `createWidgets` method and then we created a new callback method, which our new button calls when it is clicked. We placed this code just below the callback of our first button.

Our next feature request is to add more functionality. The business logic is encapsulated in another Python module. We invoke this new functionality by adding three more buttons to **Tab 1** in `GUI_Complexity_start_add_three_more_buttons.py`. We use a loop to do this.

Next, our customers ask for more features and we use the same approach in `GUI_Complexity_start_add_three_more_buttons_add_more.py`.



This is not too bad. When we get new feature requests for another 50 new features, we start to wonder whether our approach is still the best one to use.

One way to manage the increasing complexity our GUI handles is by adding tabs. By adding more tabs and placing related features into their own tab, we get control of the complexity and make our GUI more intuitive. We do this in `GUI_Complexity_end_tab3.py`, which creates our new **Tab 3**.

We saw how to handle complexity by modularizing our GUI by breaking large features into smaller pieces and arranging them in functionally related areas using tabs.

While complexity has many aspects, modularizing and refactoring the code is usually a very good approach to handling software code complexity.

GUI design using multiple notebooks

In this recipe, we will create our GUI using multiple notebooks. Surprisingly, `tkinter` does not ship out of the box with this functionality, but we can design such a widget ourselves.

Using multiple notebooks will further reduce the complexity discussed in the previous recipe.

Getting ready

We will create a new Python GUI similar to the one in the previous recipe. This time, however, we will design our GUI with two notebooks. To focus on this feature, we will use functions instead of class methods. Reading the previous recipe will be a good introduction to this recipe.

How to do it...

Let's see how to perform this recipe:

1. To use multiple notebooks within the same GUI, we start by creating two frames. The first frame will hold the notebooks and their tabs while the second frame will serve as the display area for the widgets each tab is designed to display.
2. Create a new module: `GUI_Complexity_end_tab3_multiple_notebooks.py`.
3. Add the following code:

```
import tkinter as tk
from tkinter import ttk
from tkinter import scrolledtext
from tkinter import Menu
from tkinter import Spinbox
from tkinter.messagebox import showinfo
```

4. Create callback functions:

```
def clickMe(button, name, number):
    button.configure(text='Hello {} {}'.format(name.get(),
                                                number.get()))

def clearScrol(scr):
    scr.delete('1.0', tk.END)

def _spin(spin, scr):
    value = spin.get()
    print(value)
    scr.insert(tk.INSERT, value + '\n')

def checkCallback(*ignoredArgs):
    pass
#-----
def create_display_area():
    # add empty label for spacing
    display_area_label = tk.Label(display_area, text="", height=2)
```

```
    display_area_label.grid(column=0, row=0)
#-----
def clear_display_area():
    # remove previous widget(s) from display_area:
    for widget in display_area.grid_slaves():
        if int(widget.grid_info()["row"]) == 0:
            widget.grid_forget()
#-----
def _quit():
    win.quit()
    win.destroy()
    exit()
```

5. Create a menu bar:

```
def create_menu():
    menuBar = Menu(win_frame_multi_row_tabs)
    win.config(menu=menuBar)

    fileMenu = Menu(menuBar, tearoff=0)
    fileMenu.add_command(label="New")
    fileMenu.add_separator()
    fileMenu.add_command(label="Exit", command=_quit)
    menuBar.add_cascade(label="File", menu=fileMenu)

    helpMenu = Menu(menuBar, tearoff=0)
    helpMenu.add_command(label="About")
    menuBar.add_cascade(label="Help", menu=helpMenu)
```

6. Create Tab Display Area 1:

```
def display_tab1():
    monty = ttk.LabelFrame(display_area, text=' Mighty Python ')
    monty.grid(column=0, row=0, padx=8, pady=4)

    ttk.Label(monty, text="Enter a name:").grid(column=0, row=0,
                                                sticky='W')

    name = tk.StringVar()
    nameEntered = ttk.Entry(monty, width=12, textvariable=name)
    nameEntered.grid(column=0, row=1, sticky='W')
    ttk.Label(monty, text="Choose a number:").grid(column=1, row=0)
    number = tk.StringVar()
    numberChosen = ttk.Combobox(monty, width=12,
                                textvariable=number)
    numberChosen['values'] = (1, 2, 4, 42, 100)
    numberChosen.grid(column=1, row=1)
    numberChosen.current(0)
```

```
action = ttk.Button(monty, text="Click Me!",  
command= lambda: clickMe(action, name, number))  
action.grid(column=2, row=1)  
  
scrolW = 30; scrolH = 3  
scr = scrolledtext.ScrolledText(monty, width=scrolW,  
height=scrolH, wrap=tk.WORD)  
scr.grid(column=0, row=3, sticky='WE', columnspan=3)  
  
spin = Spinbox(monty, values=(1, 2, 4, 42, 100), width=5, bd=8,  
command= lambda: _spin(spin, scr))  
spin.grid(column=0, row=2, sticky='W')  
  
clear = ttk.Button(monty, text="Clear Text", command= lambda:  
clearScrol(scr))  
clear.grid(column=2, row=2)  
  
startRow = 4  
for idx in range(12):  
    if idx < 2: col = idx  
    else: col += 1  
    if not idx % 3:  
        startRow += 1  
        col = 0  
    b = ttk.Button(monty, text="Feature " + str(idx+1))  
    b.grid(column=col, row=startRow)
```

7. Create Tab Display Area 2:

```
def display_tab2():  
    monty2 = ttk.LabelFrame(display_area, text=' Holy Grail ')  
    monty2.grid(column=0, row=0, padx=8, pady=4)  
  
    chVarDis = tk.IntVar()  
    check1 = tk.Checkbutton(monty2, text="Disabled",  
variable=chVarDis, state='disabled')  
    check1.select()  
    check1.grid(column=0, row=0, sticky=tk.W)  
    chVarUn = tk.IntVar()  
    check2 = tk.Checkbutton(monty2, text="UnChecked",  
variable=chVarUn)  
    check2.deselect()  
    check2.grid(column=1, row=0, sticky=tk.W )  
    chVarEn = tk.IntVar()  
    check3 = tk.Checkbutton(monty2, text="Toggle",  
variable=chVarEn)  
    check3.deselect()  
    check3.grid(column=2, row=0, sticky=tk.W)
```

```

labelsFrame = ttk.LabelFrame(monty2,
text=' Labels in a Frame ')
labelsFrame.grid(column=0, row=7)

ttk.Label(labelsFrame, text="Label1").grid(column=0, row=0)
ttk.Label(labelsFrame, text="Label2").grid(column=0, row=1)

for child in labelsFrame.winfo_children():
    child.grid_configure(padx=8)

```

8. Create Tab Display Area 3:

```

def display_tab3():
    monty3 = ttk.LabelFrame(display_area, text=' New Features ')
    monty3.grid(column=0, row=0, padx=8, pady=4)

    startRow = 4
    for idx in range(24):
        if idx < 2: col = idx
        else: col += 1
        if not idx % 3:
            startRow += 1
            col = 0
        b = ttk.Button(monty3, text="Feature " + str(idx + 1))
        b.grid(column=col, row=startRow)

    for child in monty3.winfo_children():
        child.grid_configure(padx=8)

```

9. Write the code to display a button for all other tabs:

```

def display_button(active_notebook, tab_no):
    btn = ttk.Button(display_area, text=active_notebook + ' - Tab ' +
    tab_no, \ command= lambda: showinfo("Tab Display",
    "Tab: " + tab_no) )
    btn.grid(column=0, row=0, padx=8, pady=8)

```

10. Create the notebook callback:

```

def notebook_callback(event):
    clear_display_area()
    current_notebook = str(event.widget)
    tab_no = str(event.widget.index("current") + 1)
    if current_notebook.endswith('notebook'):
        active_notebook = 'Notebook 1'
    elif current_notebook.endswith('notebook2'):
        active_notebook = 'Notebook 2'
    else:

```

```
        active_notebook = ''
if active_notebook is 'Notebook 1':
    if tab_no == '1': display_tab1()
    elif tab_no == '2': display_tab2()
    elif tab_no == '3': display_tab3()
    else: display_button(active_notebook, tab_no)
else:
    display_button(active_notebook, tab_no)
```

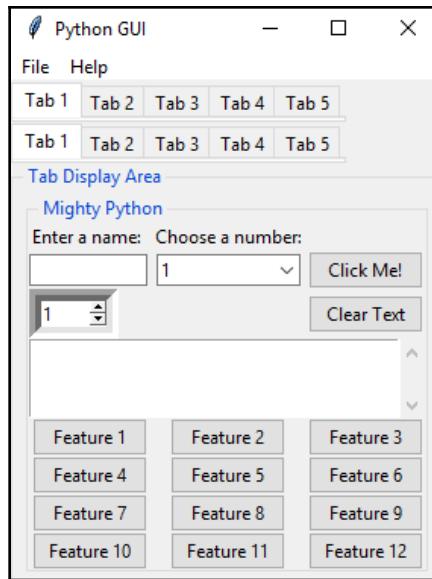
11. Create the GUI with the multiple notebooks:

```
win = tk.Tk()                      # Create instance
win.title("Python GUI")            # Add title
#-----
win_frame_multi_row_tabs = ttk.Frame(win)
win_frame_multi_row_tabs.grid(column=0, row=0, sticky='W')
display_area = ttk.Labelframe(win, text=' Tab Display Area ')
display_area.grid(column=0, row=1, sticky='WE')
note1 = ttk.Notebook(win_frame_multi_row_tabs)
note1.grid(column=0, row=0)
note2 = ttk.Notebook(win_frame_multi_row_tabs)
note2.grid(column=0, row=1)
# create and add tabs to Notebooks
for tab_no in range(5):
    tab1 = ttk.Frame(note1, width=0, height=0)
    # Create a tab for notebook 1
    tab2 = ttk.Frame(note2, width=0, height=0)
    # Create a tab for notebook 2
    note1.add(tab1, text=' Tab {} '.format(tab_no + 1))
    # Add tab notebook 1
    note2.add(tab2, text=' Tab {} '.format(tab_no + 1))
    # Add tab notebook 2

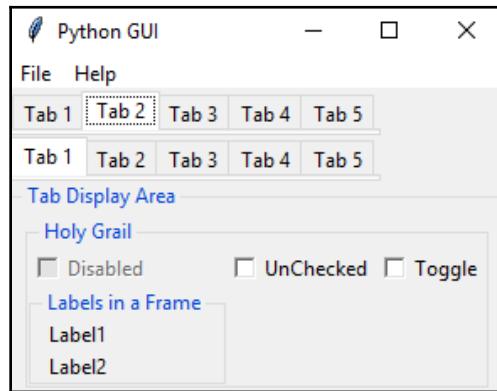
# bind click-events to Notebooks
note1.bind("<ButtonRelease-1>", notebook_callback)
note2.bind("<ButtonRelease-1>", notebook_callback)

create_display_area()
create_menu()
display_tab1()
#-----
win.mainloop()
#-----
```

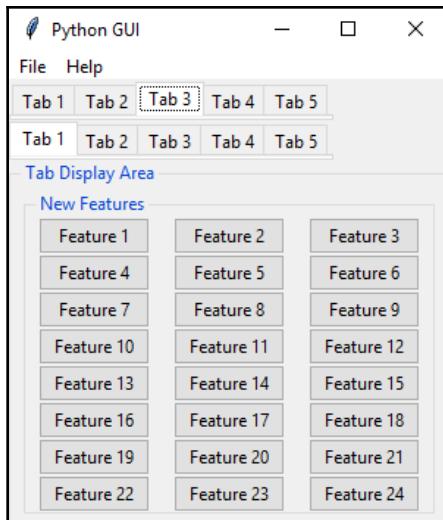
12. Run the code, click on **Tab 1**, and observe the following output:



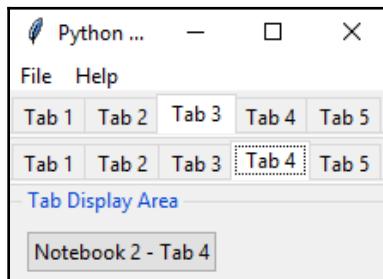
13. Click on **Tab 2**. You will see the following output:



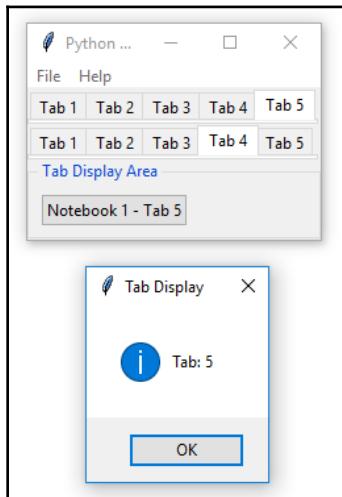
14. Click on **Tab 3**. You will see the following output:



15. Click on **Tab 4** in the second row and observe the following output:



16. Click on **Tab 5** in the first row and then click the button in **Tab Display Area** to see the following output:



Let's go behind the scenes to understand the code better.

How it works...

In `GUI_Complexity_end_tab3_multiple_notebooks.py`, we use the grid layout manager to arrange the two frames we are creating, placing one above the other. Then, we create two notebooks and arrange them within the first frame:

```
④ #-----
#  Create GUI
#-----
win = tk.Tk()          # Create instance
win.title("Python, GUI") # Add title
#-----

win_frame_multi_row_tabs = ttk.Frame(win)
win_frame_multi_row_tabs.grid(column=0, row=0, sticky='W')

display_area = ttk.LabelFrame(win, text=" Tab Display Area ")
display_area.grid(column=0, row=1, sticky='WE')

note1 = ttk.Notebook(win_frame_multi_row_tabs)
note1.grid(column=0, row=0)

note2 = ttk.Notebook(win_frame_multi_row_tabs)
note2.grid(column=0, row=1)
```

Next, we use a loop to create five tabs and add them to each notebook:

```
# create and add tabs to Notebooks
for tab_no in range(5):
    tab1 = ttk.Frame(note1, width=0, height=0)           # Create a tab for notebook 1
    tab2 = ttk.Frame(note2, width=0, height=0)           # Create a tab for notebook 2
    note1.add(tab1, text=' Tab {} '.format(tab_no + 1)) # Add tab notebook 1
    note2.add(tab2, text=' Tab {} '.format(tab_no + 1)) # Add tab notebook 2
```

We create a callback function and bind the click event of the two notebooks to this callback function. Now, when the user clicks on any tab belonging to the two notebooks, this callback function will be called:

```
# bind click-events to Notebooks
note1.bind("<ButtonRelease-1>", notebook_callback)
note2.bind("<ButtonRelease-1>", notebook_callback)
```

In the callback function, we add logic that decides which widgets get displayed after clicking a tab:

```
-----  
@def notebook_callback(event):
    clear_display_area()

    current_notebook = str(event.widget)
    tab_no = str(event.widget.index("current")) + 1

    if current_notebook.endswith('notebook'):
        active_notebook = 'Notebook 1'
    elif current_notebook.endswith('notebook2'):
        active_notebook = 'Notebook 2'
    else:
        active_notebook = ''

    if active_notebook is 'Notebook 1':
        if tab_no == '1': display_tab1()
        elif tab_no == '2': display_tab2()
        elif tab_no == '3': display_tab3()
        else: display_button(active_notebook, tab_no)
    else:
        display_button(active_notebook, tab_no)
```

We add a function that creates a display area and another function that clears the area:

```
#-----#
@def create_display_area():
    # add empty label for spacing
    display_area_label = tk.Label(display_area, text="", height=2)
    display_area_label.grid(column=0, row=0)

#-----#
@def clear_display_area():
    # remove previous widget(s) from display_area:
    for widget in display_area.grid_slaves():
        if int(widget.grid_info()["row"]) == 0:
            widget.grid_forget()
```



Note how the `notebook_callback()` function calls the `clear_display_area()` function.

The `clear_display_area()` function knows both the row and column in which the widgets of tabs are being created, and, by finding row 0, we can then use `grid_forget()` to clear the display.

For tabs 1 to 3 of the first notebook, we create new frames to hold more widgets. Clicking any of those three tabs then results in a GUI very similar to the one we created in the previous recipe.

These first three tabs are being invoked in the callback function as `display_tab1()`, `display_tab2()`, and `display_tab3()` when those tabs are being clicked.

Here is the code that runs when clicking on **Tab 3** of the first notebook:

```
#-----
@def display_tab3():
    monty3 = ttk.LabelFrame(display_area, text=' New Features ')
    monty3.grid(column=0, row=0, padx=8, pady=4)

    # Adding more Feature Buttons
    startRow = 4
    for idx in range(24):
        if idx < 2:
            colIdx = idx
            col = colIdx
        else:
            col += 1
        if not idx % 3:
            startRow += 1
            col = 0

        b = ttk.Button(monty3, text="Feature " + str(idx + 1))
        b.grid(column=col, row=startRow)

    # Add some space around each label
    for child in monty3.winfo_children():
        child.grid_configure(padx=8)
```

Clicking any tab other than the first three tabs of the first notebook one calls the same function, `display_button()`, which results in a button being displayed whose text property is being set to show the notebook and tab number:

```
#-----
@def display_button(active_notebook, tab_no):
    btn = ttk.Button(display_area, text=active_notebook + ' - Tab ' + tab_no, \
                     command= lambda: showinfo("Tab Display", "Tab: " + tab_no) )
    btn.grid(column=0, row=0, padx=8, pady=8)
```

Clicking any of these buttons results in a message box.

At the end of the code, we invoke the `display_tab1()` function. When the GUI first starts up, the widgets of this tab are what get displayed in the display area:

```
# bind click-events to Notebooks
note1.bind("<ButtonRelease-1>", notebook_callback)
note2.bind("<ButtonRelease-1>", notebook_callback)

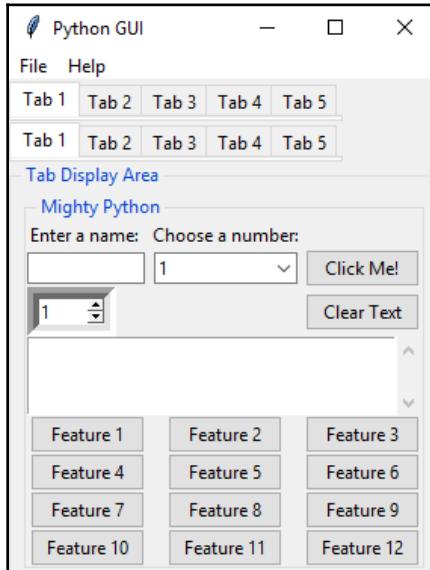
create_display_area()

create_menu()

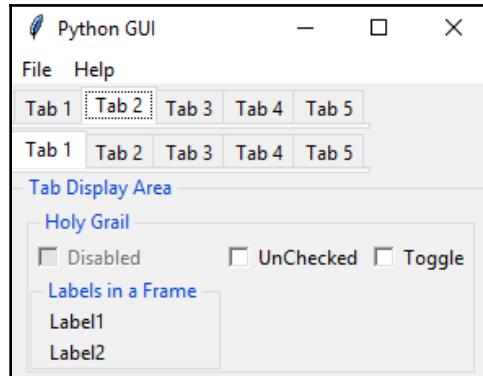
display_tab1()

#-----
win.mainloop()
#-----
```

Running the `GUI_Complexity_end_tab3_multiple_notebooks.py` code of this recipe creates the following GUI:



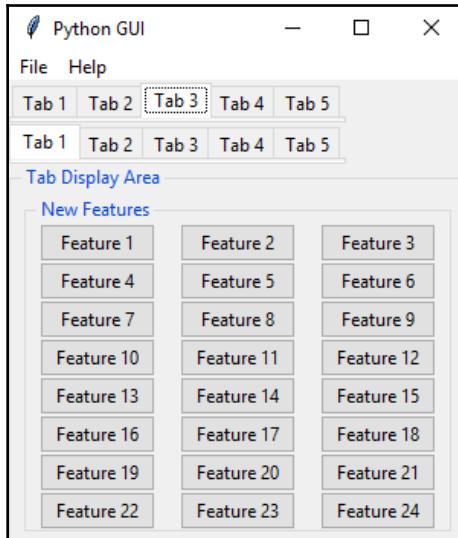
Clicking on **Tab 2** of the first notebook clears the tab display area and then displays the widgets created in the `display_tab2()` function:



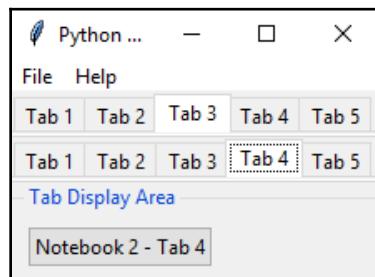


Note how the tab display area automatically adjusts to the sizes of the widgets being created.

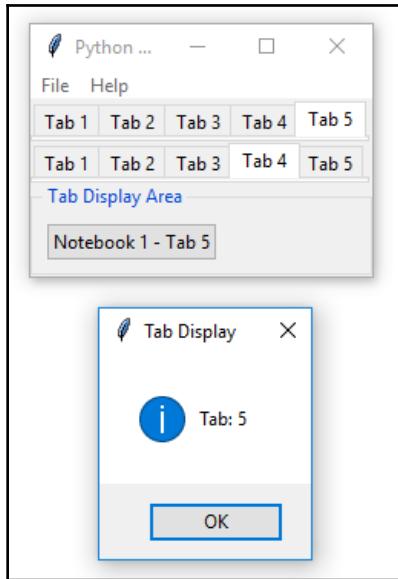
Clicking **Tab 3** results in the following GUI display:



Clicking any other tab in either the first or the second notebook results in a button being displayed in the tab display area:



Clicking any of those buttons results in a message box:



There is no limit to creating notebooks. We can create as many notebooks as our design requires.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

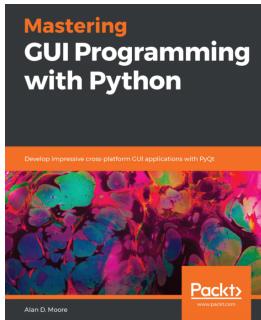


Learn Python by Building Data Science Applications

Philipp Kats, David Katz

ISBN: 978-1-78953-536-5

- Code in Python using Jupyter and VS Code
- Explore the basics of coding – loops, variables, functions, and classes
- Deploy continuous integration with Git, Bash, and DVC
- Get to grips with Pandas, NumPy, and scikit-learn
- Perform data visualization with Matplotlib, Altair, and Databricks
- Create a package out of your code using poetry and test it with PyTest
- Make your machine learning model accessible to anyone with the web API



Mastering GUI Programming with Python

Alan D. Moore

ISBN: 978-1-78961-290-5

- Get to grips with the inner workings of PyQt5
- Learn how elements in a GUI application communicate with signals and slots
- Learn techniques for styling an application
- Explore database-driven applications with the QtSQL module
- Create 2D graphics with QPainter
- Delve into 3D graphics with QOpenGLWidget
- Build network and web-aware applications with QtNetwork and QtWebEngine

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

- .py code
 - Designer .ui code, converting into 368, 369, 370
- .whl extension
 - installation, using pip with .whl extension 144
 - Matplotlib, installing with pip 144
 - wheel module, installing 147, 148, 149
 - wheel module, using 145
- - __init__ to connect modules
 - using 406, 407, 408, 409, 410
 - working 410, 411
- A**
 - agile fashion
 - GUI, designing 281, 282, 283, 284
- B**
 - bugs 288
 - buttons
 - creating 18, 19, 20, 21
 - text attributes, modifying 18, 19, 20, 21
- C**
 - callback functions
 - working 139
 - writing 137, 138
 - Canvas 114
 - canvas widget
 - third tab, creating in GUI 115
 - using 113, 114, 115
 - working 115
 - chart
 - add-ons 153
 - adjusting, to range of data 169, 170
 - creating 159
 - creating, with Matplotlib 151
 - data lines, plotting 159, 160, 161, 163
 - data, creating in Python module 167, 168, 169
 - data, creating with Python lists 167
 - graph, plotting 163, 164
 - labels, placing 153
 - legend, adding 159, 164
 - matplotlib.pyplot, importing 152
 - plotting 159
 - scale, adjusting dynamically 167
 - scaling 164, 165, 166, 167
 - check button widgets
 - creating, with different initial states 31, 32, 33, 34
 - code naming convention
 - using 416, 417
 - working 418
 - coding
 - in classes 132, 133, 134, 136, 137
 - combobox widgets
 - creating 27, 28, 29
 - state attribute, passing into constructor 30, 31
 - communicate.py code
 - executing 342, 344
 - complexity
 - avoiding 428, 430, 432, 434, 435, 436, 437
 - constraint 251
 - controls
 - adding, with wxPython 321
 - converted Designer code 371, 372, 373
 - Coordinated Universal Time (UTC) 275
 - Create, Read, Update, and Delete (CRUD) 218

D

daemon 183
data
 reading, from websites with urlopen 212, 213, 214, 216
 retrieving, from MySQL database 252, 253, 256, 257
 storing, from MySQL database 252, 253, 256, 257
debug output levels
 configuring 292, 293, 294, 295, 296
debug watches
 setting up 289, 290, 291, 292
design patterns
 using 425, 426, 427
 working 427, 428
Designer .ui code
 converting, into .py code 368, 369, 370
Designer tool
 buttons, adding 388, 389, 390, 391, 392, 393, 394, 395
 labels, adding 388, 389, 390, 391, 392, 393, 394, 395
 layouts, adding 388
 layouts, using 386, 387, 388
 used, for adding Tab Widget to UI 383, 384, 385, 386
dialog widgets
 used, for copying files to network 198, 201, 202, 204, 205, 206, 207, 208
Don't Repeat Yourself (DRY principle) 37, 141

E

Eastern Daylight Time (EDT) 276
Eclipse PyDev IDE
 used, for writing unit tests 306, 307, 308, 309, 310, 312
Exit menu item
 functionality, connecting to 379, 380, 381, 382, 383

F

fall-down coding style
 mixing, with OOP coding style 413, 415, 416

files

 copying, with dialog widgets to network 198, 201, 202, 204, 205, 206, 207, 208
First In, First Out (FIFO) 186
focus() method 24
functionality
 connecting, to Exit menu item 379, 380, 381, 382, 383

G

Graphical User Interfaces, with Tk
 reference link 10
grid layout manager
 using 80, 81
 working 81, 82
GUI code
 testing 284, 285, 286, 287, 288
GUI design
 creating, with multiple notebooks 437, 438, 439, 441, 442, 444, 445, 447, 449, 451
GUI form
 label, adding 16, 17, 18
GUI language
 modifying 268, 269, 271, 272
GUI using widgets
 expanding 54, 56, 57, 58, 59
GUI widgets
 aligning, by embedding frames within frames 59, 60, 62, 63, 64
GUI, for internationalization
 preparing 276, 277, 278, 279, 280, 281

GUI

 designing, in agile fashion 281, 282, 283, 284
 localizing 272, 273, 274, 275, 276
 Progressbar, adding 109, 110, 111, 112
 title, modifying 352, 353, 354

I

Integrated Development Environments (IDEs) 289

L

label frame widget
 labels, arranging within 46, 47, 48, 49
labels
 adding, to GUI form 16, 17, 18

placing, on chart 153
Last In, First Out (LIFO) 186
lazy initialization design pattern 207
loop
 widgets, adding 41, 42

M

main root window
 icon, modifying 95, 96
Matplotlib chart
 creating 151, 152, 153, 154, 155, 156, 157
Matplotlib
 download link 147
 installation, using pip with .whl extension 144
 installing, in chart 151
 used, for creating chart 151
Matplotlib_labels.py
 working 158
menu bar widget
 adding 358, 359
menu bars
 creating 64, 65, 66, 67, 68, 69, 70
 menu item, adding 376, 377, 378, 379
 working 71, 72
menu item
 adding, to menu bar 376, 377, 378, 379
message boxes
 callback function, adding 89
 creating, in Python 85, 86, 87, 88, 89
 error, creating 84
 Help functionality, adding 84
 information, creating 84
 warning, creating 84
modular GUI design
 building 374, 375, 376
module-level global variables
 using 126, 127, 128, 129, 130
 working 131
MS Visual C++ Build Tools, from Stack Overflow
 installation link 150
multiple threads
 creating 174, 175, 176, 177
 working 177
MySQL database connection
 configuring 225, 226, 227, 228, 229

MySQL database
 data, retrieving from 252, 253, 256, 257
 data, storing from 252, 253, 256, 257
MySQL server
 connecting, from Python 219, 221, 223, 224
 installing, from Python 219, 221, 223, 224
MySQL Workbench
 download link 257
 using 257, 259, 262, 263

N

network
 dialog widgets, used for copying files 198, 201, 202, 204, 205, 206, 207, 208
 TCP/IP, used to communicate via 208, 209, 210, 211

O

object-oriented programming (OOP)
 about 12
 avoiding 420, 421, 422, 424
 code, refactoring 354, 355, 356
OOP coding style
 mixing, with fall-down coding style 412, 413, 415, 416
orphan records 251

P

padding
 using, to add space around widgets 49, 50, 51, 52, 53, 54
pip
 used, for installing Matplotlib with .whl extension 144
Progressbar
 adding, to GUI 109, 110, 111, 112
 working 112, 113
PyQt5 actions
 creating 359
PyQt5 Designer form
 saving 366, 367, 368
PyQt5 Designer tool
 installing 349, 350
 working with 360, 361, 362, 363
PyQt5 Designer

form, previewing 363, 365, 366
PyQt5 GUI framework
 working with 395
PyQt5 GUI
 writing 351, 352
PyQt5
 installation link 347
 installing 347, 348, 349
Python code
 writing, to communicate GUIs 339
Python exception handling
 reference link 216
Python GUI database
 designing 230, 231, 232, 233, 234, 235, 236,
 237, 238, 239
Python GUI
 creating 10, 11, 14
 creating, in wxPython 317
 preventing, from resized 13, 14, 15
 working 12, 13, 15, 16
Python's `__main__` section
 used, for creating self-testing code 296, 297,
 299, 300, 301
Python
 message boxes, creating 85, 86, 87, 88, 89
 MySQL server, connecting from 219, 221, 223,
 224
 MySQL server, installing from 219, 221, 223,
 224
 used, for creating tooltips 105, 106, 107, 108
 using, to control tkinter GUI frameworks 334,
 337, 338
 using, to control wxPython GUI frameworks 334,
 337, 338
 writing, to communicate GUIs 341

Q

QMainWindow
 inheriting from 356, 357
queues
 passing, to different modules 193, 194, 196,
 197
 using 186, 187, 188, 189, 190, 191, 192

R

radio button widgets
 symbolic color names 38
 using 34, 35, 36, 37
regression 288
relationships 242
relief attribute
 applying 102, 103, 104, 105
reusable GUI components
 creating 139, 141
robust GUIs
 creating, with unit tests 301, 302, 303, 304,
 305, 306

S

SCHEMAS 263
scrolled text widgets
 creating 38, 39
 using 38, 40, 41
self-testing code
 creating, with Python's `__main__` section 296,
 297, 299, 300, 301
semicolon 237
separation of concerns (SoC) 268, 374
spaghetti code
 avoiding 399, 400, 401, 402
 working 403, 404, 405
spin box control
 using 97, 98, 99, 100, 101
SQL DELETE command
 using 247, 248, 250, 251, 252
SQL INSERT command
 using 239, 240, 241, 242
SQL UPDATE command
 using 242, 243, 244, 245, 246, 247
stateoverflow
 reference link 327
status bar widget
 adding 357, 358
StringVar() type
 using 117, 118, 120, 122
 working 123, 124
Structured Query Language (SQL) 217
subplot

reference link 158
symbolic color names
reference link 38

T

Tab Widget
adding, to UI with Designer tool 383, 384, 385, 386
tabbed widgets
creating 72, 73, 74, 75, 77, 79
working 79
Tcl/Tk
reference link 10
Test Fixtures
about 307
reference links 307
Test-Driven-Development (TDD) methodology 305
textbox widgets
about 21
creating 21, 22
working 22, 23
thread
stop() method, using 183, 185, 186
working 178, 180, 181, 182
tkinter app
wxPython app, embedding 327, 328, 329, 330
tkinter GUI code
embedding, into wxPython 331, 332, 333
tkinter GUI frameworks
Python, using 334, 337, 338
tkinter message boxes
arguments, passing 94
creating 89, 90, 91, 92, 93
tkinter protocol
reference link 163
tkinter window form
built-in tkinter attribute, using 95
title, creating 94
tkinter.ttk
reference link 17
tooltips
creating, with Python 105, 106, 107, 108
Transmission Control Protocol/Internet Protocol (TCP/IP)
about 208

using, to communicate via network 208, 209, 210, 211

Ttk Notebook widget
reference link 79

U

unit tests
used, for creating robust GUIs 301, 302, 303, 304, 305, 306
writing, with Eclipse PyDev IDE 306, 307, 308, 309, 310, 312
Universal Naming Convention (UNC) 207
urlopen
used, for reading data from websites 212, 213, 214, 216

W

websites
urlopen, used for reading data 212, 213, 215, 216
widget text
displaying, in languages 265, 266, 267, 268
widgets
adding, in loop 41, 42
data, obtaining from 124, 125, 126
disabling 24, 25, 26, 27
focus, setting 24, 25, 26, 27
working 43
WinMerge 236
wxPython app
embedding, in tkinter app 327, 328, 329, 330
working 330
wxPython GUI frameworks
Python, using 334, 337, 338
wxPython GUI toolkit
reference link 317
wxPython library
installing 314
pip, using to install wxPython framework 316
using, with Python 3.7 315, 316
wxPython Phoenix version
reference link 315
wxPython
module, installing 322
Python GUI, creating 317

Python module, creating 326, 327
reference link 315
tkinter GUI code, embedding 331, 332, 333

URL 316
used, for adding controls 321, 322, 323, 325
working 321
working window, creating 318, 319, 320