Beginning PyQt

A Hands-on Approach to GUI Programming

Joshua M. Willman

Beginning PyQt

A Hands-on Approach to GUI Programming

Joshua M. Willman

Apress®

Beginning PyQt: A Hands-on Approach to GUI Programming

Joshua M. Willman
Hampton, VA, USA

To my daughter, Kalani.
Pursue what inspires you.

Table of Contents

About the Author

Joshua M. Willman began using Python in 2015, when his first task was to build neural networks using machine learning libraries, including Keras and TensorFlow, for image classification. While creating large image datasets for his research, he needed to build a GUI that would simplify the workload and labeling process, which introduced him to PyQt. He currently works as a Python developer and instructor, designing courses to help others learn about coding in Python for game development, AI and machine learning, and programming using microcontrollers. More recently, he set up the site RedHuli to explore his and others' interests in using Python and programming for creative purposes.

About the Technical Reviewer

Lentin Joseph is an author and robotics entrepreneur from India. He runs a robotics software company called Qbotics Labs in India. He has 9 years of experience in the robotics domain primarily in ROS, OpenCV, and PCL.

He has authored eight books in ROS, namely, *Learning Robotics Using Python* 1st and 2nd edition, *Mastering ROS for Robotics Programming* 1st and 2nd edition, *ROS Robotics Projects*, *Robot Operating System for Absolute Beginners*, and *ROS Programming*.

He pursed a master's degree in robotics from India and also did research at Robotics Institute, CMU, USA. He is also a TEDx speaker.

Acknowledgments

Writing this book has truly been an eye-opening experience, and I owe thanks to those who have helped me reach this point in my life.

I want to begin by thanking those individuals who helped me to take a good, hard look at my life.

To Joyce Corriere, your lessons and kindness all those years ago have inspired me to this day.

Thank you to Professor Rong Xiong (熊蓉) for giving me the chance to pursue my dreams and to get my life back on track.

I owe a debt of gratitude to Lingyan Xu (许凌雁) and Melody for their laughs and support during some of the toughest times.

When I first started learning Python and PyQt a few years back, I had no idea where to begin. A big thanks to the Python community and to Jan Bodnar at ZetCode for giving me the tools to get started in creating my own applications.

Thank you so much to Apress editorial for the serendipitous e-mail that began this whole journey just as I was researching ways to write a book.

A very immense thank you must be given to my Coordinating Editor, Divya Modi, for being patient and positively stoic with me all of the times the deadlines flew past.

Thanks to my mother, Valorie, and my sisters, Teesha and Jazzmin, for all of the support you have given me.

Words cannot express how deeply grateful I am to my wife, Evelyn, who has been the most patient of all during this time, listening to my incessant rambling about ideas, and who helped me realize that I cannot put everything into one book.

To Kalani, your laughs have been an enormous uplift.

Lastly, thank you to the readers. I hope the ideas found within this book can help you in some way.

Introduction

Just getting started is more important than anything else. Coding a graphical user interface (GUI) can be thought of as a combination between programming and graphic design skills. An awareness of a user's needs is crucial for both usability and graphical appearance. Programming a GUI is often a matter of finding the right component, referred to as widgets, to complete a task, and then applying the necessary programming skills to make them operational.

In this book, we will see how to use the Python programming language, along with the PyQt5 toolkit, to create GUIs. With PyQt5, many of the components are already created for you. However, if you ever find yourself needing a component that does not exist, with PyQt5 you can always make your own custom widgets and classes, as well.

If this is your first time creating GUIs at all, then my recommendation is to follow along with **Chapters 2 through 6** to get your bearings with PyQt. Many of the key concepts and classes that you will use for basic interfaces can be found there, including creating a window for arranging widgets, making components that are interactive and can communicate with one another and with the information stored on your computer, layout management, setting up the menu system, and manipulating a GUI's appearance.

In the remaining chapters, we will begin looking at more specific examples: looking at Qt Designer for simplifying the GUI design process in **Chapter 7**; using the clipboard in **Chapter 8**; art and animation and creating your own widgets in **Chapter 9**; working with databases in **Chapter 10**; threading in **Chapter 11**; and a number of miscellaneous topics in **Chapter 12**.

There are also two appendices, **Appendix A** which gives extra information about PyQt classes and **Appendix B** for refreshing your knowledge about Python.

No one chapter has all the tools you will need in it. Widgets and classes are spread throughout the book, helping you to learn and apply what you have learned as you go. Nor is every one of PyQt's classes covered within these pages. Learning is an ongoing process, and sometimes having to do a bit of extra searching will help ideas better stick in your mind.

CHAPTER 1

Charting the Course

Hello! Welcome to *Beginning PyQt: A Hands-on Approach to GUI Programming*. The goal of this book is to take a more practical approach to learning how to code user interfaces (UIs), following along and coding numerous examples, both simple and complex, to help understand and visualize how we can use the concepts taught in each chapter. What that means is that when you learn how to code QPushButton widgets, for example, you will first walk through a simple program that helps you build the fundamentals. Then, you will apply that concept to a slightly larger project.

"When am I ever going to use this?" I can still recall sitting in my math classes and hearing someone ask that question. The formulas and theories, culminations of numerous mathematicians' life's work, were all amazing to learn, but without some way to apply them to actual examples beyond the textbook, those concepts faded away into some dark recess in my mind.

To avoid spiraling down this same path when learning to code, this book aims to help you jump right into actual examples to get you coding and practicing the concepts with a hands-on approach. New concepts and PyQt classes are introduced in each chapter, and later chapters sometimes build upon the previous ones. Of course, not everyone has the same goal in mind. Therefore, there are a couple of ways to approach the content of this book. The first way is for readers who want to follow along and practice learning many of the basics of PyQt. These types of readers are encouraged to code many of the projects and then play around with the concepts to design their own applications. The other approach is for those readers who already have a project in mind and need some help getting started. You are definitely encouraged to use the code in this book as a foundation to build your own projects and get them off the ground.

Who Should Read This Book

Everyone must begin somewhere. With that idea in mind, this book is targeted for individuals who already have a fundamental understanding of the Python programming language and are looking to either expand their skills in Python or have a project where they need to create a UI, but may have no prior experience creating UI or no idea where to begin. Having prior knowledge of other Python UI toolkits is not necessary to get started in this book.

Introduction to User Interfaces

The user interface (UI) has become a key component of our everyday lives, becoming the intermediary between us and our ever-growing number of machines. A UI is designed to facilitate in human-computer interaction. The human needs to operate and control the machine to serve some purpose; meanwhile the machine needs to simultaneously provide feedback to aid the human's decision-making process. UIs are everywhere, from the mobile applications on our phones to web browsers, to heavy machinery controls, and even on the appliances in our kitchens. Of course, the ways in which we interact with technology are not merely limited to our hands, as many UIs also allow interaction with our other sensory organs.

A good UI is tasked with helping a person produce a desired result while also allowing for easier, more efficient, and more friendly operation of a machine. Think about the photo editing apps on your phone. Editing the size, color, or exposure is practically effortless as you slide your fingers across the screen and watch the images change almost instantly. The user provides minimal input to achieve the desired output.

What Is a Graphical User Interface?

For this book, we will be focusing on creating **graphical user interfaces** (**GUIs**) which take advantage of a computer's graphics capabilities to create visual controls on a machine's screen. This makes interaction with machines much easier. Decades ago, users would have to use the command line and text commands to interact with the computer. Tasks such as opening, deleting, and moving files or searching through directories were all done by typing in certain commands. However, these were not very user friendly or simple to use. So GUIs were created to allow users to interact with electronic devices using graphical controls, rather than command-line interfaces.

These graphical control elements, or **widgets**, such as buttons, menus, and windows, make such tasks effortless. Interaction now becomes as simple as moving your mouse or touching the screen depending upon your device and clicking the widget.

Concepts for Creating Good Interface Design

This, first and foremost, is a technical book written to help those of you who want to learn how to create and code your own GUI with PyQt and Python. That being said, if you plan to design any kind of UI that other people will use, then you are no longer creating a UI just to solve some problem. You must also begin to consider other users of the application, as well. Think about what you want them to accomplish, or how the application can help them. Sometimes when we are trying to solve a problem, we get so caught up in trying to create a product that we forget about the people who actually have to interact with them.

The following are a list of guidelines to consider when designing your own UI. They are not set rules and by no means a complete list, but rather ideas that can help to save you some time and headaches later.

1. Clarity – Using clear language, hierarchy, and flow with visual elements to avoid ambiguity. One of the ways this can be achieved is by considering visual importance to the human eye, laying out widgets with bigger sizes, darker colors, and so on in such a manner that we can visually understand the UI.

2. Conciseness – Simplifying the layout to include only what the user needs to see or interact with at a given time in order to be brief, but also comprehensive. Adding more labels or buttons in your window just to give the user more options is not always better.

3. Consistency – Design the UI so that there is consistency across the application. This helps users to recognize patterns in the visual elements and layout and can be seen in typography that improves the navigation and readability of the application, image styles, or even color schemes.

4. Efficiency – Utilizing good design and shortcuts to help the user improve productivity. If a task can be accomplished in two steps, why design it so that it has to be completed in five?

5. Familiarity – Consider elements that users normally see in other UIs and how they would expect them to perform in your applications. For example, think about how weird it would be to have to enter your login information and the password entry field is above the username. It is not wrong, but now you are unnecessarily making users think about their actions and slowing them down.

6. Responsive – Give the user feedback, for example, a toggle that changes color to "on" or "off," a small message to notify the user if their input is correct or incorrect, or even a sound effect to verify a completed action. The user should never be left wondering if their action was successful or not.

The PyQt Framework

The PyQt application is a set of Python 2 and Python 3 bindings for the Qt cross-platform widget toolkit and application framework. What does that mean?

First, Qt is used for the development of graphical user interfaces and other applications and is currently being developed by The Qt Company. The framework is significant because it can run on numerous software and hardware systems such as Windows, MacOS, Linux, Android, or embedded systems with little to no change to the underlying code and is still able to maintain capabilities and speed of the system on which it is being run.

Second, this all means that PyQt combines all the advantages of the Qt C++ cross-platform widget toolkit with Python, the powerful and simple, cross-platform interpreted language.

For more information about PyQt, check out

www.riverbankcomputing.com/news.

Why Choose PyQt?

PyQt is capable of more than just creating GUIs, as it also has access to Qt classes that cover mechanics such as XML handling, SQL databases, network communication, graphics and animations, and many other technologies. Take the capabilities of Qt and combine it with the number of extension modules that Python provides, and you have the ability to create new applications that can build upon these preexisting libraries.

PyQt also includes Qt Designer, which allows for anyone to create a GUI much faster using a simple drag and drop graphical interface designer.

Using PyQt's signal and slot mechanism, you can essentially create your own widgets that can call other Python functions. This will be covered in more detail in Chapter 9.

There are, of course, other toolkits available for creating applications with GUIs using Python, such as Tkinter or wxPython. The many other toolkits have some advantages over PyQt. For example, Tkinter comes bundled with Python, meaning that you can find an abundance of helpful resources by doing a quick search on the Internet.

It is worth noting that if you choose to use PyQt to create commercial applications, you may need to get a license.

Ultimately, it all comes down to choosing the toolkit that works the best for your project.

Requirements

In order to use PyQt, you will first need to have **Python 3** installed. To check if Python is already installed on your system or to find out how to download Python, please refer to Appendix B. You will also find a guide to help you refresh your Python skills to aid you while learning PyQt in Appendix B.

Note   As of this writing, Python 2 is set to no longer be maintained. Therefore, all Python code in this book will be written using Python 3. Many of the projects that utilize Python 2 have already started making their way over to Python 3. If you have any questions about the differences between 2 and 3, I suggest checking out http://python-future.org/compatible_idioms.html.

PyQt does not come included with your Python installation. For this book we will be using the PyQt5 toolkit, which is the latest version. Please refer to Appendix A to learn how to download PyQt for your operating system.

Links to Source Code

The source code for *Beginning PyQt: A Hands-on Approach to GUI Programming* can be found on GitHub via the book's product page, located at www.apress.com/ISBN.

How This Book Is Organized

In the beginning chapters, we will walk through the code step-by-step, helping to guide you through PyQt classes and concepts for designing GUIs. **Chapters** 2 and 3 will help you get started using PyQt, adding more and more functionality to your projects. Each chapter teaches how to use different widgets, such as QLabel, QCheckBox, and QLineEdit, and gives examples and ideas of how to use them. **Chapter** 3 will also introduce you to PyQt's signals and slots mechanism for handling events.

Chapter 4 focuses on layout managers for arranging widgets. After learning about different widgets, **Chapter** 5 guides you through examples that help you to create menus and toolbars. **Chapter** 6 presents style sheets for altering the look of your applications and how to reimplement event handlers.

Since Qt also includes its own graphical user interface to help you create GUIs, we will take a look at how to use Qt Designer in **Chapter** 7.

Chapters 8 **through** 11 begin looking at larger concepts and projects, including using the clipboard to move between applications, graphics, and animation, creating custom widgets, utilizing SQL databases and PyQt's model/view architecture, and multithreading programming.

Chapter 12 contains extra example projects to help you continue to gain extra practice and insight into creating applications with PyQt.

Appendix A guides you through the process of downloading PyQt5 and includes information about different PyQt classes. **Appendix B** is there to help you set up Python 3 and to refer back to in case you are not sure about some of the Python code used in this book.

Reader Feedback

Finally, your feedback, questions, and ideas are very important. If you would like to take a moment to let me know your thoughts about the book, you can send comments to the following address:

redhuli.comments@gmail.com

Getting Started with PyQt

Hello again! If this is your first time to ever make any kind of UI, then you might be wondering where to begin. Luckily in this chapter we are going to start you off with learning some of the fundamentals before jumping into the heavy lifting.

Creating any kind of UI can seem like a formidable task with all the different layouts, windows, and widgets there are to consider. Widgets are the buttons, menus, sliders, and other components that will make up our user interfaces. Therefore in every chapter, we are going to discuss how to build one or more projects and break them down into incremental steps and tasks in order to better help you understand the larger programs.

For example, in this chapter we are going to be looking at how to create a user profile GUI in PyQt. First, we will discuss what a user profile is generally comprised of and what kinds of widgets we will need for this project. Then, we will go step-by-step in learning how to build a basic window and add images and text to the GUI. Once it's all done, you'll have an application that looks like Figure 2-1.

Figure 2-1. *User profile that displays your information for others to view*

Note   For those who already have a project in mind or just need to learn how to include certain widgets in their applications, many of the chapters will include smaller programs focused specifically on how to code them.

Project 2.1 – User Profile GUI

A user profile refers to some kind of visual display used to present a specific user's personal data. The data on the profile helps to associate certain characteristics with that user and allows others to collect information about that individual. User profiles can be found on a number of different platforms including computer programs, online social networking sites, such as LinkedIn or Facebook, or operating systems. Depending upon what environment you are looking at, the appearance of the profile will change to fit the goals, rules, and needs of that application.

User profiles often have a number of parameters which are either mandatory or optional and allow for some level of customization to fit the preferences of the user, for example, a profile image or background colors. Many of them contain features which can typically be found in all types of profiles, such as the user's name and the "About" section to share some information about the user.

For this project, you will

1. Create an empty window in PyQt and find out

   a. About the basic classes and modules needed to set up your GUI

   b. How to modify the window size and title

2. Learn about creating widgets

   a. Specifically QLabel to add text and images to your GUI

   b. How to organize the widgets in your window using move()

Design the GUI Layout

Let's break down the user profile GUI and plan out what widgets you need and how they will be arranged in the window as seen in Figure 2-2.
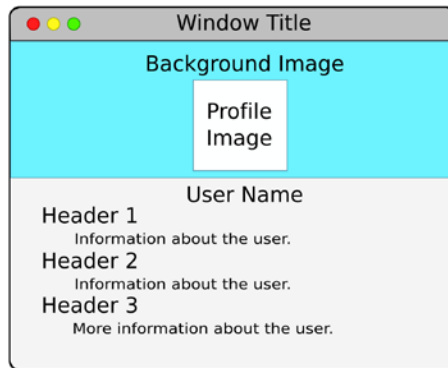
Figure 2-2. *Layout for the user profile GUI*

In real user profile applications, there are often a combination of different widgets. Some are interactive, allowing the viewer to click the links in the profile or "Like" buttons, while others are only meant to be read and cannot be altered by the viewer. In this chapter, we are looking at designing a basic interface in PyQt that shows a way to display information in the window using the **QLabel** widget.

The user interface can be divided into two parts:

1. The background image and profile image on the top.

2. And the user's name and information on the bottom. The text on the bottom can further be broken down into smaller sections that are delineated by the use of different font sizes.

Create an Empty Window

A GUI application generally consists of a main window and possibly one or more **dialog boxes**. The main window in your program can consist of a menubar, a status bar, and other widgets, whereas a dialog is made up of buttons and is created to communicate information to the user and prompt them for input. An alert window that pops up asking you if you want to save changes to your document is an example of a dialog. Dialog boxes will be covered further in Chapter 3.

Listing 2-1 walks you through the steps to create an empty GUI window.

Listing 2-1. Create an empty window in PyQt

```python
# basic_window.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QWidget

class EmptyWindow(QWidget):
    def __init__(self):
        super().__init__() # create default constructor for QWidget
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
```

```
        self.setGeometry(100, 100, 400, 300)
        self.setWindowTitle('Empty Window in PyQt')
        self.show()

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = EmptyWindow()
    sys.exit(app.exec_())
```

Your initial window should look similar to the one in Figure 2-3 depending upon your operating system.
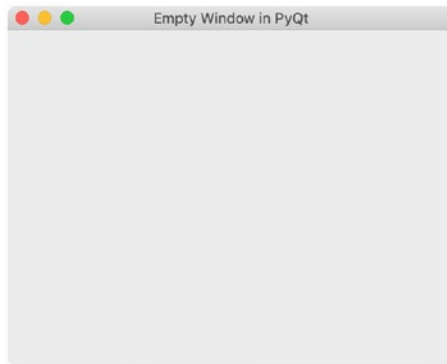
Figure 2-3. *Empty window created with PyQt5*

Explanation

Walking through the code, we first start by importing the sys and PyQt5 modules that we need to create a window. We commonly use sys in order to pass command-line arguments to our applications and to close them.

The QtWidgets module provides a set of UI elements that can be used to create desktop-style GUIs. From the QtWidgets module, we import two classes, QApplication and QWidget. You only need to create a single instance of the QApplication class, which manages the application's main event loop, flow, initialization, and finalization, as well as session management. Take a quick look at

```
    app = QApplication(sys.argv)
```

11

QApplication takes as an argument sys.argv. You can also pass in an empty list if you know that your program will not be taking any command-line arguments using

```
 app = QApplication([])
```

Next we create a window object that inherits from the class we created, EmptyWindow. Our class actually inherits from QWidget, which is the base class for which all other user interface objects are derived.

We need to call the show() method on the window object to display it to the screen. This is located inside the initializeUI() function in our EmptyWindow class. You will notice app.exec_() in the final line of the program. This function starts the event loop and will remain here until you quit the application. sys.exit() ensures a clean exit.

If all of this is a little confusing as to why we have to create an application before we create the window, think of QApplication as the frame that contains our window. The window, our GUI, is created using QWidget. Before we can create our GUI, we must create an instance of QApplication that we can place window in. Take a look at the following code to better see the order of creating a window in PyQt5 using procedural programming:

```
# 1. Import necessary modules
import sys # use sys to accept command-line arguments
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication(sys.argv) # 2. Create application object
window = QWidget() # 3. Create window
window.show() # 4. Call show to view GUI
sys.exit(app.exec_()) # Start the event loop and use sys.exit # to close
the application
```

Modifying the Window

The preceding EmptyWindow class contains a function, initializeUI(), that creates the window based upon the parameters we specify. The initializeUI() function is reproduced as follows:

```
    def initializeUI(self):
        """

        Initialize the window and display its contents to the screen.
```

```
        """
        self.setGeometry(100, 100, 400, 300)
        self.setWindowTitle('Empty Window in PyQt')
        self.show()
```

setGeometry() defines the location of the window on your computer screen and its dimensions, width and height. So the window we just started is located at x=100, y=100 in the window and has width=400 and height=300. setWindowTitle() is used to change the title of our window.

We will look at further customization of the window's layout in Chapter 4 and appearance in Chapter 6.

Note   Throughout the book we will be looking at Python code written using **object-oriented programming (OOP)**. If you need a refresher on OOP, there is a quick guide that can be found in Appendix B.

The QLabel Widget

Now that we have a fundamental understanding of what it takes to create the window, we can move on and add more functionality with widgets using **QLabel**. A QLabel object acts as a noneditable placeholder to display text, images, or movies. It is also useful for creating labels around other widgets to specify their roles or give them titles. QLabel widgets can also display plain text, hyperlinks, or rich text.

In Listing 2-2 we are going to take a look at extending the ability of our window by showing how to create both text and image labels.

Listing 2-2. Create an empty window in PyQt

```
# labels.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QLabel
from PyQt5.QtGui import QPixmap

class HelloWorldWindow(QWidget):
```

```python
    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 250, 250)
        self.setWindowTitle('QLabel Example')
        self.displayLabels()

        self.show()

    def displayLabels(self):
        """
        Display text and images using QLabels.

        Check to see if image files exist, if not throw an
exception.
        """
        text = QLabel(self)
        text.setText("Hello")
        text.move(105, 15)

        image = "images/world.png"
        try:
            with open(image):
                world_image = QLabel(self)
                pixmap = QPixmap(image)
                world_image.setPixmap(pixmap)
                world_image.move(25, 40)
        except FileNotFoundError:
            print("Image not found.")
```

```
# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = HelloWorldWindow()
    sys.exit(app.exec_())
```

Once you run the program, you should see a window similar to Figure 2-4.
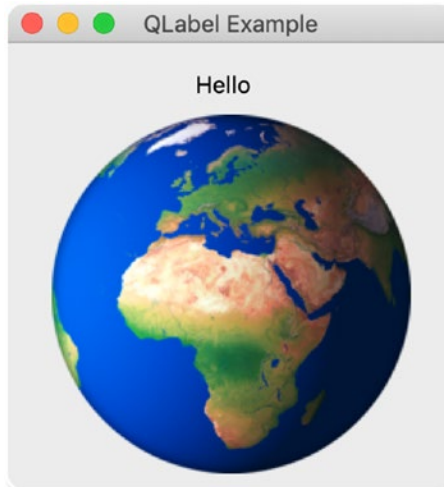
Figure 2-4. *Example of using QLabel widgets to create images and text*

Explanation

We begin again by first importing the necessary PyQt modules. To create the window, we need to import the QtWidgets class, and since this time we are going to be using the QLabel widget, we need to include it as well in our import statement.

This time we also need to import the QtGui module, as well. QtGui handles numerous graphical elements. QPixmap is a Qt class that is optimized for showing images on the screen.

We then go through a similar process of creating our application, creating a HelloWorldWindow class that inherits from the QWidget base class, initializing the size of the window with setGeometry() and the title of our GUI. Then we use the show() method to display the window and use exec_() to begin the event loop. Finally, sys. exit() is used to close our program.

The `HelloWorldWindow` class contains a `displayLabels()` function that we will use to display text and images. First, you must create a `QLabel` object and specify what the label will say using `setText()`. Here the text is set to "Hello". In the following line, we use the `move()` function to arrange the label in the window:

```
text = QLabel(self)
text.setText("Hello")
text.move(105, 15)
```

PyQt5 has a number of layout methods including horizontal layouts, grid layouts, as well as **absolute positioning**. For the programs created in this chapter, we will be using absolute positioning with the `move()` method. With `move()`, you only need to specify the x and y pixel values of the widget's top-left corner to arrange it in the window. For our text label, we specify the values to be x=105 and y=15.

Our image is loaded in a similar fashion, creating a `QLabel` to be placed in the main window. Then we construct a `QPixmap` of the image and use `setPixmap()` to show the image displayed on the `world_image` label. The label's absolute location is set using `move()`.

Each of PyQt's different classes has their own methods that can be used to customize and change their look and functionality. In Appendix B, you can find a list of the widgets used in this book along with some of the more common methods you are likely to use to modify them.

User Profile GUI Solution

You have now learned many of the basic tools used to create the user profile GUI. This project is comprised of all QLabel widgets and its goal is to help you learn the fundamentals of creating GUIs in PyQt5. The QLabel widgets are used to display personal information specified by the user.

After you have followed along with Listing 2-3, it is encouraged to practice modifying the size of the window, add your own text or images, and practice using other QLabel methods to see how all of the different parts work together to make the GUI window.

Listing 2-3. Code for the user profile GUI

```python
# user_profile.py
# Import necessary modules
import sys, os.path
from PyQt5.QtWidgets import QApplication, QLabel, QWidget
from PyQt5.QtGui import QFont, QPixmap

class UserProfile(QWidget):
    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(50, 50, 250, 400)
        self.setWindowTitle("2.1 - User Profile GUI")
        self.displayImages()
        self.displayUserInfo()

        self.show()

    def displayImages(self):
        """
        Display background and profile images.

        Check to see if image files exist, if not throw an exception.
        """
        background_image = "images/skyblue.png"
        profile_image = "images/profile_image.png"

        try:
            with open(background_image):
                background = QLabel(self)
                pixmap = QPixmap(background_image)
                background.setPixmap(pixmap)
        except FileNotFoundError:
            print("Image not found.")
```

```python
        try:
            with open(profile_image):
                user_image = QLabel(self)
                pixmap = QPixmap(profile_image)
                user_image.setPixmap(pixmap)
                user_image.move(80, 20)
        except FileNotFoundError:
            print("Image not found.")

    def displayUserInfo(self):
        """
        Create the labels to be displayed for the User Profile.
        """
        user_name = QLabel(self)
        user_name.setText("John Doe")
        user_name.move(85, 140)
        user_name.setFont(QFont('Arial', 20))

        bio_title = QLabel(self)
        bio_title.setText("Biography")
        bio_title.move(15, 170)
        bio_title.setFont(QFont('Arial', 17))

        about = QLabel(self)
        about.setText("I'm a Software Engineer with 8 years\
            experience creating awesome code.")
        about.setWordWrap(True)
        about.move(15, 190)

        skills_title = QLabel(self)
        skills_title.setText("Skills")
        skills_title.move(15, 240)
        skills_title.setFont(QFont('Arial', 17))

        skills = QLabel(self)
        skills.setText("Python  |  PHP  |  SQL  |  JavaScript")
        skills.move(15, 260)
```

```python
        experience_title = QLabel(self)
        experience_title.setText("Experience")
        experience_title.move(15, 290)
        experience_title.setFont(QFont('Arial', 17))

        experience = QLabel(self)
        experience.setText("Python Developer")
        experience.move(15, 310)

        dates = QLabel(self)
        dates.setText("Mar 2011 - Present")
        dates.move(15, 330)
        dates.setFont(QFont('Arial', 10))

        experience = QLabel(self)
        experience.setText("Pizza Delivery Driver")
        experience.move(15, 350)

        dates = QLabel(self)
        dates.setText("Aug 2015 - Dec 2017")
        dates.move(15, 370)
        dates.setFont(QFont('Arial', 10))

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = UserProfile()
    sys.exit(app.exec_())
```

Explanation

If you followed along in this chapter, then this project is just a much longer version containing a background image and more text labels. Images for the labels are loaded using Python try-except clauses.

A few new things here are as follows:

- We import the `QFont` class from the `QtGui` module, allowing us to modify the size and types of fonts in our application using QLabel's `setFont()` method.

- Using `move()`, we are able to easily overlap images. Take a look at the `displayImages()` function to see how to do so using absolute positioning.

Summary

At this point, you should have a fundamental understanding for getting started in creating your own GUIs with PyQt5. We looked at setting up a basic window with QApplication and QWidget classes, made some simple modifications to the look of the window, learned how to create text and image labels with QLabel, and saw how to arrange them in the window using the `move()` method. In subsequent chapters, we will continue to learn about more widgets and classes and learn how to use them to fit the requirements of the applications we wish to create.

It is worth noting that this user profile is by no means complete. A user's profile is generally more interactive, including links, buttons, and menus. As you go through other chapters, you should come back and improve the user profile GUI as a means to apply what you have learned.

CHAPTER 3

Adding More Functionality to Interfaces

In Chapter 2, we took a look at how to get started in PyQt, set up the main window, and learned how to create and arrange multiple QLabel widgets to create a simple application. However, none of what we did was very interactive. What good is a user interface if all you can do is stare at it?

You're in luck because this chapter is all about setting you on the path to making interfaces that are more interactive and responsive. We will take a look at some new fundamental widgets that will help us to build our next project, a functional login GUI. To make things clearer and easier for you to follow along, the login GUI will be divided into two parts, the actual login interface and a new user registration window.

Before we get started, let's take a look at the new widgets and useful concepts that will be covered in this chapter.

1. Learn about new kinds of widgets and classes, including

   a. `QPushButton` – One of the most common widgets for giving our computer simple commands

   b. `QLineEdit` – Which gives the user fields to input information

   c. `QCheckBox` – Which can act as a binary switch

   d. `QMessageBox` – Useful for displaying alert or information dialog boxes

2. Find out about event handling with `Signals` and `Slots` in PyQt.

3. Understand the differences between windows and dialog boxes when creating UIs.

Project 3.1 – Login GUI

While it might not seem like much, the login GUI, or the login screen, is probably one of the most common interfaces you interact with on a regular basis. Signing into your computer, your online bank account, e-mail, or social media accounts, logging into your phone, or signing up for some new app, the login GUI is everywhere.

The login GUI can appear to be quite a simple user interface. However, it is actually very complex for a number of reasons. First of all, it acts as the interface that allows us to access our own personal data. You want to create a GUI that clearly labels its widgets, differentiates between where to sign in and where to register a new account, and helps users to better navigate through potential errors, such as if caps lock is on or if the username is incorrect. Secondly, the appearance of the login GUI and methods in which we log in to our devices have changed dramatically over the years, allowing users to log in using Touch ID or their social media accounts. This means that there is no single design that will work for every platform.

For this project, we are going to focus on creating a simple login UI that

- Allows the user to enter their username and password and calls a function to check if their information matches one that is stored in a text file

- Displays appropriate messages depending upon whether login is successful, if an error has occurred, or if we simply want to close the window

- Displays or hides the password by clicking a checkbox

- Allows the user to create a new account by clicking a "sign up" button that will open a new window

---

Note   There are two projects in this chapter, the `login GUI` and the `create new user GUI`. They are actually one entire project that has been separated into two parts to make it easier for you to follow along, or for those who only need one part of the project and not the other.

After following along with this chapter, you will be able to make a login GUI like the one seen in Figure 3-1.

Figure 3-1. *Simple login GUI*

Design the Login GUI

While the look and layout of the login GUI may change between platforms, they generally have a few key components that are common throughout, such as

- Username and password entry fields

- Checkboxes that may remember the user's login information or reveal the password

- Buttons that users can click to log in or even register for a new account

For this project, we will focus on trying to implement most of those features. (The "remember me" checkbox that is common in a lot of login GUIs is beyond the scope of this chapter as it involves using cookies or working with PyQt's QSettings class.)

The layout for our login GUI can be seen in Figure 3-2. For this project, we will need to create a few QLabels to help users understand the purpose of this application and to give titles to our username and password entry fields.
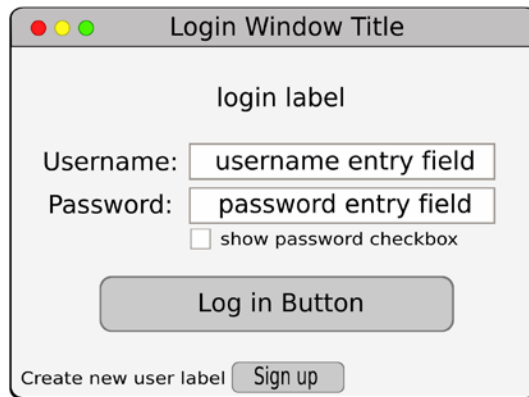
Figure 3-2. *Layout for login GUI*

For the areas where users will enter their information, we create two separate QLineEdit widgets. Under the password line edit widget, there is a checkbox that the user can check if they want to view or hide the password they entered.

There are two QPushButtons, one that the user can click to log in and the other to register a new account. When the user clicks the login button, we will create a function that is called to check if the user exists. If the user information is correct, we will display a QMessageBox which tells the user that login is successful. Otherwise, another QMessageBox is displayed to alert the user to an error.

If the user's information does not exist, they can click the sign up button and a new window will appear where they can enter their information. This part is covered in the section "Project 3.2 – Create New User GUI."

Finally, we will learn how to change the event handler for when the user closes the window. Rather than just closing the application, we will first display a dialog box that will confirm whether or not the user really wants to quit.

The QPushButton Widget

Let's first take a look at a fundamental widget that you will probably use in almost every GUI you create, **QPushButton**. The QPushButton can be used to command the computer to perform some kind of operation or answer a question. When you click the QPushButton widget, it sends out a signal that can be connected to a function. Common buttons you might encounter are OK, Next, Cancel, Close, Yes, and No.

Buttons are typically displayed with either a text label or an icon that describes its action or purpose. There are a number of different kinds of buttons with different usages that can be created including QToolButtons and QRadioButtons.

For our first example, we are going to take a look at how to set up a QPushButton that, when clicked, will call a function that closes our application (Listing 3-1).

Listing 3-1. Code for learning how to add QPushButton widgets to your application

```python
# button.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QLabel, QPushButton

class ButtonWindow(QWidget):
    def __init__(self):
        super().__init__() # create default constructor for QWidget
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 200, 150)
        self.setWindowTitle('QPushButton Widget')
        self.displayButton() # call our displayButton function

        self.show()

    def displayButton(self):
        '''
        Setup the button widget.
        '''
        name_label = QLabel(self)
        name_label.setText("Don't push the button.")
        name_label.move(60, 30) # arrange label

        button = QPushButton('Push Me', self)
        button.clicked.connect(self.buttonClicked)
```

```
        button.move(80, 70) # arrange button

    def buttonClicked(self):
        '''
        Print message to the terminal,
        and close the window when button is clicked.
        '''
        print("The window has been closed.")
        self.close()

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = ButtonWindow()
    sys.exit(app.exec_())
```

When you finish, your window should look similar to Figure 3-3.

Figure 3-3. *Example of the QPushButton widget*

Explanation

We begin by importing sys and the necessary PyQt classes including QApplication and QWidget for creating our application object and window, respectively. For this program we will also import the QLabel and QPushButton widgets which are also part of the QtWidgets module.

Next let's create our own ButtonWindow class which inherits from QWidget. Here we will initialize the window and widgets we need for our GUI. The ButtonWindow class has

two functions, `displayButton` and `buttonClicked`. In the `displayButton` function, we create a label and set its text using `setText()`. If you look at the portion of code where the `button` is created, we set the text on the button as a parameter of `QPushButton`. We could also set the text as follows:

```
button.setText("Don't push the button.")
```

When you click the `QPushButton` with your mouse, it will send out the signal `clicked()`. After we create the button, use

```
button.clicked.connect(self.buttonClicked)
```

to connect the signal to the action we want the button to perform, in this case `self.buttonClicked`. A QPushButton widget can also be set to be activated by the spacebar or using a keyboard shortcut. The `buttonClicked` function calls `self.close()` to close the application.

Note   In the preceding example, the signal `clicked()` is connected to our function. There are also other kinds of signals that the `QPushButton` can send out including `pressed()` when the button is down, `released()` when the button is released, or `toggled()` that can be used like a binary switch.

Events, Signals, and Slots

Before we go on, you should be introduced to an important concept when building GUI applications in PyQt. GUIs are **event-driven**, meaning that they respond to events that are created by the user, from the keyboard or the mouse, or by events caused by the system, such as a timer or when connecting to Bluetooth. No matter how they are generated, the application needs to listen for these events and respond to them in some way, also known as **event handling**. For example, when `exec_()` is called, the application begins listening for events until it is closed.

In PyQt, event handling is performed with signals and slots. **Signals** are the events that occur when a widget's state changes, such as when a button is clicked or a checkbox is toggled on or off. Those signals then need to be handled in some way. **Slots** are the methods that are executed in response to the signal. Slots are simply Python functions or built-in PyQt functions that are connected to an event and executed when the signal occurs.

Take a look at the following code from the earlier QPushButton program:

```
button.clicked.connect(self.buttonClicked)
```

When we push on the button, a clicked() signal is emitted. In order to make use of that signal, we must connect() to some callable function, in this case buttonClicked(), which is the slot.

Put simply, widgets send out signals and we collect and use them with slots to make our application perform some action.

Many widgets have predefined signals and slots, meaning you only need to call them in order to get the behavior you want for your application.

The topic of signals and slots and how to make some custom signals will be covered in more detail and with examples in Chapter 6.

The QLineEdit Widget

The next widget we are going to take a look at is the **QLineEdit** widget. For our login GUI, we need to create areas where the user can input the text for their username and password on a single line. QLineEdit also supports normal text editing functions such as cut, copy and paste, and redo or undo if you need to add those features to your program.

The QLineEdit widget also has a number of methods to add more functionality to your GUI, such as hiding text when it is entered, using placeholder text, or even setting a limit on the length of the text that can be input.

In Listing 3-2 we will take a look at how to set up the QLineEdit widget, retrieve the text using the text() function, and see how to clear the text that the user inputs.

Note   If you need multiple lines to enter text in, use QTextEdit.

Listing 3-2. Code for learning how to add QLineEdit widgets to your application

```
# lineedit.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget,
    QLabel, QLineEdit, QPushButton)
from PyQt5.QtCore import Qt
```

```python
class EntryWindow(QWidget): # Inherits QWidget

    def __init__(self): # Constructor
        super().__init__() # Initializer which calls constructor for QWidget
        self.initializeUI() # Call function used to set up window

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 400, 200)
        self.setWindowTitle('QLineEdit Widget')
        self.displayWidgets()

        self.show()

    def displayWidgets(self):
        '''
        Setup the QLineEdit and other widgets.
        '''
        # Create name label and line edit widgets
        QLabel("Please enter your name below.", self).move(100, 10)
        name_label = QLabel("Name:", self)
        name_label.move(70, 50)

        self.name_entry = QLineEdit(self)
        self.name_entry.setAlignment(Qt.AlignLeft) # The default alignment
        is AlignLeft
        self.name_entry.move(130, 50)
        self.name_entry.resize(200, 20) # Change size of entry field

        self.clear_button = QPushButton('Clear', self)
        self.clear_button.clicked.connect(self.clearEntries)
        self.clear_button.move(160, 110)

    def clearEntries(self):
        '''
        If button is pressed, clear the line edit input field.
```

```
        '''
        sender = self.sender()
        if sender.text() == 'Clear':
            self.name_entry.clear()

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = EntryWindow()
    sys.exit(app.exec_())
```

Take a look at Figure 3-4 to get an idea of how your GUI should look.
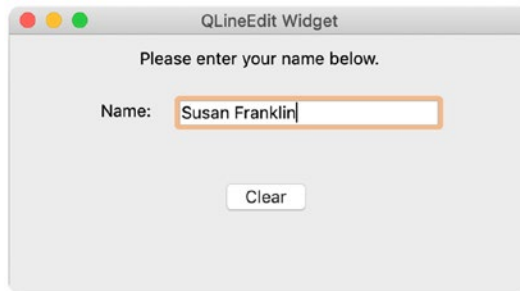
Figure 3-4. *Example of how to use QLineEdit and QPushButton widgets*

Explanation

The user can enter their name into the QLineEdit widget and click the "Clear"
QPushButton to clear their text. Other features could include clearing multiple widgets'
states when the button is clicked or checking to make sure the text entered fits the
guidelines you need in your application.

We begin by importing the necessary widgets, this time making sure to include the
QLineEdit widget which is a member of the QtWidgets module. We also import Qt from
the QtCore module. Qt contains various miscellaneous methods for creating GUIs. After
initializing our window in the EntryWindow class, the displayWidgets() function is
called that sets up the label, line edit and button widgets. When text is entered into the
name_entry widget, by default, the text starts on the left and is centered vertically.

```
self.name_entry.setAlignment(Qt.AlignLeft)
```

If you wish to change this, you could change the flag in SetAlignment from Qt.
AlignLeft to Qt.AlignRight or Qt.AlignHCenter.

When the clear_button is clicked, it emits a signal that is connected to the
clearEntries() function. In order to determine where the source of a signal is coming
from in your applications, you could also use the sender() method. Here, the signal is
sent from our button when it is clicked, and if the text on the sender is 'Clear', then the
name_entry widget reacts to the signal and clears its current text.

The QCheckBox Widget

The **QCheckBox** widget is a selectable button that generally has two states, on or off.
Since checkboxes normally have only two states, they are perfect for representing
features in your GUI that can either be enabled or disabled or for selecting from a list of
options like in a survey.

The QCheckBox can also be used for more dynamic applications, as well. For
example, you could use the checkbox to change the title of the window or even the text of
labels when enabled.

Listing 3-3 shows how to set up a window like in a questionnaire. The user is allowed
to select all checkboxes that apply to them, and each time the user clicks a box, we call a
function to show how to determine the widget's current state.

Note   The checkboxes in QCheckBox are not mutually exclusive, meaning you
can select more than one checkbox at a time. To make them mutually exclusive,
add the checkboxes to a QButtonGroup object.

Listing 3-3.   Code for learning how to add QCheckBox widgets to your
application

```
# checkboxes.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QCheckBox, QLabel)
from PyQt5.QtCore import Qt

class CheckBoxWindow(QWidget):
```

```python
    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 250, 250)
        self.setWindowTitle('QCheckBox Widget')
        self.displayCheckBoxes()

        self.show()

    def displayCheckBoxes(self):
        '''
        Setup the checkboxes and other widgets
        '''
        header_label = QLabel(self)
        header_label.setText("Which shifts can you work? (Please check all
        that apply)")
        header_label.setWordWrap(True)
        header_label.move(10, 10)
        header_label.resize(230, 60)

        # Set up checkboxes
        morning_cb = QCheckBox("Morning [8 AM-2 PM]", self) # text, parent
        morning_cb.move(20, 80)
        #morning_cb.toggle() # uncomment if you want box to start off checked,
                    # shown as an example here.
        morning_cb.stateChanged.connect(self.printToTerminal)

        after_cb = QCheckBox("Afternoon [1 PM-8 PM]", self) # text, parent
        after_cb.move(20, 100)
        after_cb.stateChanged.connect(self.printToTerminal)

        night_cb = QCheckBox("Night [7 PM-3 AM]", self) # text, parent
        night_cb.move(20, 120)
        night_cb.stateChanged.connect(self.printToTerminal)
```

```
    def printToTerminal(self, state): # pass state of checkbox
        '''
        Simple function to show how to determine the state of a checkbox.
        Prints the text label of the checkbox by determining which widget
        is sending the signal.
        '''
        sender = self.sender()
        if state == Qt.Checked:
            print("{} Selected.".format(sender.text()))
        else:
            print("{} Deselected.".format(sender.text()))

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = CheckBoxWindow()
    sys.exit(app.exec_())
```

Figure 3-5 shows our application that allows users to select multiple checkboxes.
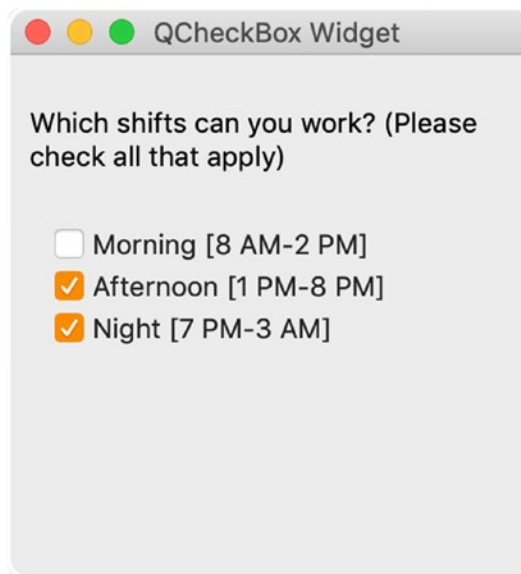
Figure 3-5. *Example of QCheckBox widgets*

Explanation

Much of this application starts off similar to before, so let's jump right into the displayCheckBoxes() method within the CheckBoxWindow class.

A QLabel widget is created so that the person looking at the window can understand the purpose of the GUI. Then three checkboxes are created, each with a variable name that is representative of the widget's purpose. Since the widgets are created in a similar manner, we will just take a look at the first one, morning_cb.

```
morning_cb = QCheckBox("Morning [8 AM-2 PM]", self)
morning_cb.move(20, 80) # arrange widget in window
#morning_cb.toggle() # uncomment if you want box to start off
checked, shown as an example here.
morning_cb.stateChanged.connect(self.printToTerminal)
```

The checkbox is created by calling the QCheckBox class and then, as parameters, assigning it text that will appear beside the actual checkbox and its parent window. The toggle() method can be used to toggle the checkbox, and uncommenting the code will cause the widget to begin as enabled when starting the program. When a checkbox's state changes, rather than using clicked() like with the QPushButton, we can use stateChanged() to send a signal and then connect to our function, printToTerminal().

The printToTerminal() function takes as a parameter state, the state of the checkbox. If a checkbox is checked, we can find out by using the isChecked() method. If the state of the button isChecked(), then use the sender() method to find out which button is sending a signal and print its text value to the terminal window. An example of the output to the terminal can be seen in Figure 3-6.

```
Morning [8 AM-2 PM] Selected.
Night [7 PM-3 AM] Selected.
Afternoon [1 PM-8 PM] Selected.
Night [7 PM-3 AM] Deselected.
Morning [8 AM-2 PM] Deselected.
Afternoon [1 PM-8 PM] Deselected.
Morning [8 AM-2 PM] Selected.
```

Figure 3-6. *Output to terminal from QCheckBox example program*

The QMessageBox Dialog Box

Often when a user is going to close an application, save their work, or an error occurs, a dialog box will pop up and display some sort of key information. The user can then interact with that dialog box, often by clicking a button to respond to the prompt.

The **QMessageBox** dialog box can not only be used to alert the user to a situation but also to allow them to decide how to handle the issue. For example, if you close a document you just modified, you might get a dialog box asking you to Save, Don't Save, or Cancel.

There are four types of predefined QMessageBox widgets in PyQt. For more details, refer to Table 3-1.

Table 3-1. *Four types of QMessageBox widgets in PyQt. Images from* `www.riverbankcomputing.com`

QMessageBox Icons	Types	Details
	Question	Ask the user a question.
	Information	Display information during normal operations.
	Warning	Report noncritical errors.
	Critical	Report critical errors.

Windows vs. Dialogs

When creating a GUI application, you will more than likely come across the terms windows and dialogs. However, windows and dialogs are not the same. Using dialog boxes in an application can make it both easier for you to develop your GUI and for the user to better understand and navigate through your application.

The **window** generally consists of menus, a toolbar, and other kinds of widgets within it that can often act as the main interface in a GUI application.

A **dialog box** will appear when the user needs to be prompted for additional information in order to continue, often to gather input such as an image or a file. After that information is given, the dialog box is normally destroyed. Dialog boxes can also be used to display options or information while a user is working in the main window. Most kinds of dialog boxes will have a parent window that will be used to determine the position of the dialog with respect to its owner. This also means that communication occurs between the window and the dialog box and allows for updates in the main window.

There are two kinds of dialog boxes, the **modal** dialog box and the **modeless** dialog box. Modal dialogs block user interaction from the rest of the program until the dialog box is closed. Modeless dialogs allow the user to interact with both the dialog and the rest of the application.

How dialog boxes appear and are used can often be influenced by the operating system you use and the guidelines set by that OS.

How to Display a QMessageBox

The QMessageBox class produces a modal dialog box, and in Listing 3-4, we will take a look at how to use two of the predefined QMessageBox message types, `Question` and `Information`.

Listing 3-4. Code for learning how to display QMessageBox dialogs

```
# dialogs.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
    QMessageBox, QLineEdit, QPushButton)
from PyQt5.QtGui import QFont

class DisplayMessageBox(QWidget):

    def __init__(self):
        super().__init__()

        self.initializeUI() # Call our function used to set up window

    def initializeUI(self):
        """
```

```
    Initialize the window and display its contents to the screen
    """
    self.setGeometry(100, 100, 400, 200)
    self.setWindowTitle('QMessageBox Example')
    self.displayWidgets()

    self.show()

def displayWidgets(self):
    """
    Set up the widgets.
    """
    catalogue_label = QLabel("Author Catalogue", self)
    catalogue_label.move(20, 20)
    catalogue_label.setFont(QFont('Arial', 20))

    auth_label = QLabel("Enter the name of the author you are searching
    for:", self)
    auth_label.move(40, 60)

    # Create author label and line edit widgets
    author_name = QLabel("Name:", self)
    author_name.move(50, 90)

    self.auth_entry = QLineEdit(self)
    self.auth_entry.move(95, 90)
    self.auth_entry.resize(240, 20)
    self.auth_entry.setPlaceholderText("firstname lastname")

    # Create search button
    search_button = QPushButton("Search", self)
    search_button.move(125, 130)
    search_button.resize(150, 40)
    search_button.clicked.connect(self.displayMessageBox)

def displayMessageBox(self):
    """
    When button is clicked, search through catalogue of names.
    If name is found, display Author Found dialog.
```

```
        Otherwise, display Author Not Found dialog.
        """
        # Check if authors.txt exists
        try:
            with open("files/authors.txt", "r") as f:
                # read each line into a list
                authors = [line.rstrip('\n') for line in f]
        except FileNotFoundError:
            print("The file cannot be found.")

        # Check for name in list
        not_found_msg = QMessageBox() # create not_found_msg object to
        avoid causing a 'referenced before assignment' error

        if self.auth_entry.text() in authors:
            QMessageBox().information(self, "Author Found", "Author found
            in catalogue!", QMessageBox.Ok, QMessageBox.Ok)
        else:
            not_found_msg = QMessageBox.question(self, "Author Not Found",
"Author not found in catalogue.\nDo you wish to continue?", QMessageBox.Yes
| QMessageBox.No, QMessageBox.No)

        if not_found_msg == QMessageBox.No:
            print("Closing application.")
            self.close()
        else:
            pass

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = DisplayMessageBox()
    sys.exit(app.exec_())
```

Figure 3-7 shows the GUI for the QMessageBox example.

Figure 3-7. *GUI to search for author's name in a text file*

Explanation

The GUI in this example consists of a few QLabel widgets, a QLineEdit widget, and a single QPushButton. For this example, you will also see how to set placeholder text in the QLineEdit widget using setPlaceholderText(). This can be helpful for a number of reasons, maybe to make the look of the window less cluttered or to give the user extra information to help them understand the format to use to input text.

The search_button sends a signal that calls the function displayMessageBox(). If the user enters a name that is contained in the authors.txt file, then an information dialog box appears like the first image in Figure 3-8. Otherwise, a question dialog box (second image in Figure 3-8) appears asking the user if they want to search again or quit the program. Let's take a look at how to create a dialog box using the QMessageBox class.

```
not_found_msg = QMessageBox.question(self, "Author Not Found", "Author
not found in catalogue.\nDo you wish to continue?", QMessageBox.Yes |
QMessageBox.No, QMessageBox.No)
```

To create a QMessageBox dialog, we first call QMessageBox and choose one of the predefined types, in this case question. Then we set the dialog title, "Author Not Found", and the text that we want to appear inside the dialog. This should inform the user about the current situation and, if necessary, notify them of actions they could take. This is followed by the types of buttons that will appear in the dialog, and each button is separated by a pipe key, |. Other types of buttons include Open, Save, Cancel, and Reset. Finally, you can specify which button you want to highlight and set as the default button.

Note   On Mac OS X, when a message box appears, the title is generally ignored due to Mac OS X Guidelines. If you are using a Mac and don't see a title in the dialog boxes, don't fear! You haven't done anything wrong.

You can also specify each of these fields in separate lines by calling `setText()`, `setWindowTitle()`, and other methods.
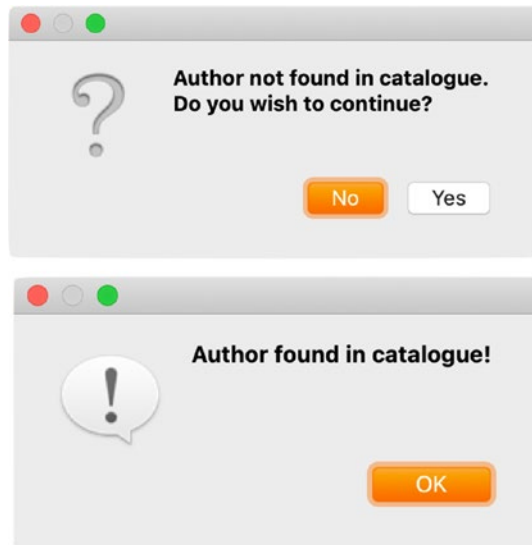
Figure 3-8. *Information dialog box (top) that lets the user know that their search was successful. However, if the author doesn't exist, a question dialog box appears asking the user to take some sort of action by clicking a button (bottom)*

Login GUI Solution

Now that we have covered the key widgets in this chapter and how to implement dialog boxes, we should have all the necessary concepts down to tackle the login GUI (Listing 3-5). Refer to Figures 3-1 and 3-2 for the look and layout of the login GUI.

Listing 3-5. Code for login GUI

```
# loginUI.py
# Import necessary modules
import sys
```

```python
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QMessageBox,
QLineEdit, QPushButton, QCheckBox)
from PyQt5.QtGui import QFont
from PyQt5.QtCore import Qt
from Registration import CreateNewUser # Import the registration module

class LoginUI(QWidget):

    def __init__(self): # Constructor
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 400, 230)
        self.setWindowTitle('3.1 - Login GUI')
        self.loginUserInterface()

        self.show()

    def loginUserInterface(self):
        """
        Create the login GUI.
        """
        login_label = QLabel(self)
        login_label.setText("login")
        login_label.move(180, 10)
        login_label.setFont(QFont('Arial', 20))

        # Username and password labels and line edit widgets
        name_label = QLabel("username:", self)
        name_label.move(30, 60)

        self.name_entry = QLineEdit(self)
        self.name_entry.move(110, 60)
        self.name_entry.resize(220, 20)
```

```python
        password_label = QLabel("password:", self)
        password_label.move(30, 90)

        self.password_entry = QLineEdit(self)
        self.password_entry.move(110, 90)
        self.password_entry.resize(220, 20)

        # Sign in push button
        sign_in_button = QPushButton('login', self)
        sign_in_button.move(100, 140)
        sign_in_button.resize(200, 40)
        sign_in_button.clicked.connect(self.clickLogin)

        # Display show password checkbox
        show_pswd_cb = QCheckBox("show password", self)
        show_pswd_cb.move(110, 115)
        show_pswd_cb.stateChanged.connect(self.showPassword)
        show_pswd_cb.toggle()
        show_pswd_cb.setChecked(False)

        # Display sign up label and push button
        not_a_member = QLabel("not a member?", self)
        not_a_member.move(70, 200)

        sign_up = QPushButton("sign up", self)
        sign_up.move(160, 195)
        sign_up.clicked.connect(self.createNewUser)

    def clickLogin(self):
        """
        When user clicks sign in button, check if username and password
        match any existing profiles in users.txt.
        If they exist, display messagebox and close program.
        If they don't, display error messagebox.
        """
        users = {} # Create empty dictionary to store user information
        # Check if users.txt exists, otherwise create new file
        try:
```

```python
        with open("files/users.txt", 'r') as f:
            for line in f:
                user_fields = line.split(" ")
                username = user_fields[0]
                password = user_fields[1].strip('\n')
                users[username] = password
    except FileNotFoundError:
        print("The file does not exist. Creating a new file.")
        f = open ("files/users.txt", "w")

    username = self.name_entry.text()
    password = self.password_entry.text()
    if (username, password) in users.items():
            QMessageBox.information(self, "Login Successful!", "Login
            Successful!", QMessageBox.Ok, QMessageBox.Ok)
            self.close() # close program
    else:
        QMessageBox.warning(self, "Error Message", "The username or
        password is incorrect.", QMessageBox.Close, QMessageBox.Close)

def showPassword(self, state):
    '''
    If checkbox is enabled, view password.
    Else, mask password so others cannot see it.
    '''
    if state == Qt.Checked:
        self.password_entry.setEchoMode(QLineEdit.Normal)
    else:
        self.password_entry.setEchoMode(QLineEdit.Password)

def createNewUser(self):
    """
    When the sign up button is clicked, open
    a new window and allow the user to create a new account.
    """
    self.create_new_user_dialog = CreateNewUser()
    self.create_new_user_dialog.show()
```

```python
    def closeEvent(self, event):
        """
        Display a QMessageBox when asking the user if they want to quit the
        program.
        """
        # Set up message box
        quit_msg = QMessageBox.question(self, "Quit Application?",
            "Are you sure you want to Quit?", QMessageBox.No | QMessageBox.Yes,
            QMessageBox.Yes)
        if quit_msg == QMessageBox.Yes:
            event.accept() # accept the event and close the application
        else:
            event.ignore() # ignore the close event

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = LoginUI()
    sys.exit(app.exec_())
```

Your GUI should look similar to the window shown in Figure 3-1.

Explanation

After importing the necessary PyQt5 modules including QtWidgets, QtGui, and QtCore, we also need to import our Registration module which will allow new users to create a new account and then return back to the login GUI to sign in. The Registration module is covered in Project 3.2 in this chapter.

In our LoginUI class that inherits from QWidget, we initialize our GUI and then create the widgets. Refer to Figure 3-2 for the layout. We create a few QLabel widgets to hold information about our GUI and labels for the QLineEdit widgets for the username and password. Widgets are arranged in the window using the move() method.

When the sign_in_button is clicked, it sends a signal that is connected to the clickLogin() method. This function opens the users.txt file (and creates one if it doesn't exist) and stores each line into a Python dictionary, with the keys being the usernames and the values of the dictionary being the passwords. The text() method is then used to retrieve the input from the two QLineEdit widgets and checks them to see

if they match any of the key/value pairs in the users dictionary. While it isn't the most practical method, this example is a very small one and demonstrates how to use simple text files with your applications. Later we will take a look at how to use SQL to search through databases in Chapter 10.

If the username and password match a key/value pair from the file, then an information QMessageBox dialog is displayed telling the user that they are successful. They can then exit the program if they wish (in an actual application at this point you would start the main window of your program). Otherwise, a warning QMessageBox is displayed if the username or password is incorrect. These two dialog boxes can be seen in Figure 3-9.
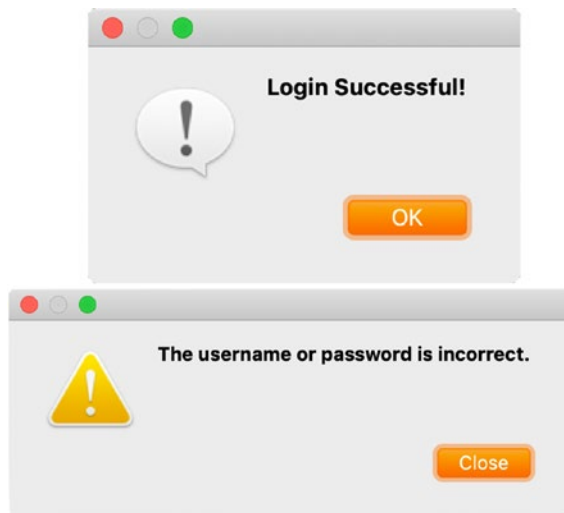
Figure 3-9. *QMessageBox dialogs that can be displayed. The information dialog box (top) lets the user know that their information was correct. The other dialog (bottom) shows a warning QMessageBox*

Hiding Input for QLineEdit

The stateChanged signal in the login UI code is connected to the showPassword() function. If the show_pswd_cb QCheckBox is checked, then the password is displayed using SetEchoMode().

```
self.password_entry.setEchoMode(QLineEdit.Normal)
```

Otherwise, if unchecked, it is hidden using

```
self.password_entry.setEchoMode(QLineEdit.Password)
```

45

If you ever need to make the text in a `QLineEdit` widget hidden from other's view, using `SetEchoMode()` can change the appearance of the text. By default, `setEchoMode()` is set to `QLineEdit.Normal`.

How to Open a New Window

If the user wants to create a new account, then they can click the `sign_up` button at the bottom of the GUI. This button sends a signal that is connected to the `createNewUser()` method which calls our `CreateNewUser` class from the `Registration` module. A new window is then opened up using `show()` where the user can enter their personal information.

Changing How the Close Event Works

Finally, currently when we want to quit our programs, we just exit by clicking the button in the top corner of the window. However, a good practice is to present a dialog box confirming whether the user really wants to quit or not. In most programs this will prevent the user from forgetting to save their latest work.

When a QWidget is closed in PyQt, it generates a `QCloseEvent`. So we need to change how the `closeEvent()` method is handled. To do so we create a new method called `closeEvent()` that accepts as a parameter an event.

In this function we create a `QMessageBox` that asks the user if they are sure about quitting. They then can click either a Yes or No button in the dialog box. We then check the value of the variable stored in `quit_msg`. If `quit_msg` is Yes, then the close `event` is accepted and the program is closed. Otherwise, the `event` is ignored.

Project 3.2 – Create New User GUI

The first time someone uses your applications you may want them to sign up and create their own username and passwords. This, of course, can allow them to personalize their accounts and then save that information for the next time they log in. The kind of information that you need from the user can range from very simple, name and gender, all the way to extremely private, Social Security numbers or bank account information. Making sure that the information that they enter is correct is very important.

Creating a New User GUI Solution

For the following project, we will have the user enter their desired username, their real name, a password, and then reenter that password to double check that it is correct. When the user clicks the sign up button, the text in the password fields will be checked for a match, and if so, the information will be saved to a text file. The user can then return to the login screen and log in.

The create new user GUI project contains many of the same widgets, including QLabel widgets, QLineEdit widgets, a QPushButton, and concepts that were part of the login UI project. Therefore, we will jump right into talking about the code shown in Listing 3-6. The completed GUI can be seen in Figure 3-10.
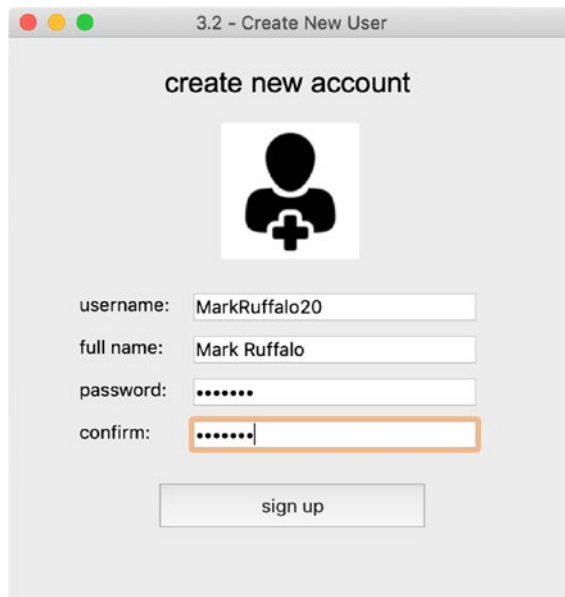
Figure 3-10. *The create new user GUI*

Listing 3-6. Code for creating a new user account GUI

```
# Registration.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QMessageBox,
QPushButton, QLabel, QLineEdit)
```

```python
from PyQt5.QtGui import QFont, QPixmap

class CreateNewUser(QWidget):

    def __init__(self):
        super().__init__()

        self.initializeUI() # Call our function used to set up window

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 400, 400)
        self.setWindowTitle('3.2 - Create New User')

        self.displayWidgetsToCollectInfo()

        self.show()

    def displayWidgetsToCollectInfo(self):
        """
        Create widgets that will be used to collect information
        from the user to create a new account.
        """
        # Create label for image
        new_user_image = "images/new_user_icon.png"
        try:
            with open(new_user_image):
                new_user = QLabel(self)
                pixmap = QPixmap(new_user_image)
                new_user.setPixmap(pixmap)
                new_user.move(150, 60)
        except FileNotFoundError:
            print("Image not found.")

        login_label = QLabel(self)
        login_label.setText("create new account")
        login_label.move(110, 20)
```

```python
login_label.setFont(QFont('Arial', 20))

# Username and fullname labels and line edit widgets
name_label = QLabel("username:", self)
name_label.move(50, 180)

self.name_entry = QLineEdit(self)
self.name_entry.move(130, 180)
self.name_entry.resize(200, 20)

name_label = QLabel("full name:", self)
name_label.move(50, 210)

name_entry = QLineEdit(self)
name_entry.move(130, 210)
name_entry.resize(200, 20)

# Create password and confirm password labels and line edit widgets
pswd_label = QLabel("password:", self)
pswd_label.move(50, 240)

self.pswd_entry = QLineEdit(self)
self.pswd_entry.setEchoMode(QLineEdit.Password)
self.pswd_entry.move(130, 240)
self.pswd_entry.resize(200, 20)

confirm_label = QLabel("confirm:", self)
confirm_label.move(50, 270)

self.confirm_entry = QLineEdit(self)
self.confirm_entry.setEchoMode(QLineEdit.Password)
self.confirm_entry.move(130, 270)
self.confirm_entry.resize(200, 20)

# Create sign up button
sign_up_button = QPushButton("sign up", self)
sign_up_button.move(100, 310)
sign_up_button.resize(200, 40)
sign_up_button.clicked.connect(self.confirmSignUp)
```

```python
    def confirmSignUp(self):
        """
        When user presses sign up, check if the passwords match.
        If they match, then save username and password text to users.txt.
        """
        pswd_text = self.pswd_entry.text()
        confirm_text = self.confirm_entry.text()

        if pswd_text != confirm_text:
            # Display messagebox if passwords don't match
            QMessageBox.warning(self, "Error Message",
                "The passwords you entered do not match. Please try
                 again.", QMessageBox.Close,
                QMessageBox.Close)
        else:
            # If passwords match, save passwords to file and return to login
            # and test if you can log in with new user information.
            with open("files/users.txt", 'a+') as f:
                f.write(self.name_entry.text() + " ")
                f.write(pswd_text + "\n")
            self.close()

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = CreateNewUser()
    sys.exit(app.exec_())
```

The completed create new user GUI can be seen in Figure 3-10.

Explanation

The create new user GUI is mainly comprised of a few QLabel widgets, four QLineEdit widgets, and a QPushButton widget that the user can click when the form is complete as can be seen in Figure 3-10.

After the user enters their information and clicks the sign_up_button, the confirmSignUp() function is called and first checks to see if the text in the pswd_entry and confirm_entry QLineEdit objects match. If they don't match, then a QMessageBox like the one in Figure 3-11 is displayed. Otherwise, the text in the name_entry and pswd_entry fields is saved to a newline in the users.txt file separated by a space which can be seen in Figure 3-12.
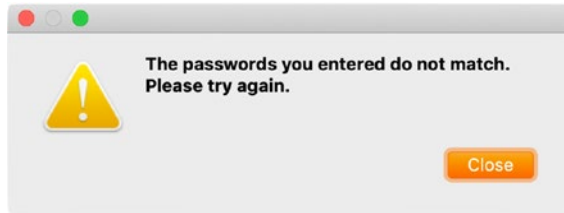
Figure 3-11. *The warning dialog displayed if the passwords you entered don't match*

If the user finishes the form and clicks the sign_up_button or closes the window before completing the form, the window will close but the login UI will still remain open. If the form was completed, the user can try to enter their new username and password into the login GUI to log in.

Figure 3-12. *The original users.txt file (top) and the updated one with a new username and a new password (bottom)*

Summary

In this chapter we took a look at some new widgets, QPushButton, QLineEdit, QCheckBox, and QMessageBox class. It is important to use dialog boxes in your program when you want the user to collect information outside of the application or to relay important details to the user. But you also should not have a dialog box pop up for every little nuance or with very little helpful information.

The applications in this chapter are by no means complete. They are the framework to get you started making your own GUIs. For example, you could make sure that the user's password includes capital and lowercase letters and other characters to make it more safe. Another possibility is to let the user know if the username they want to create already exists. Once you learn how to implement menus, you could even have the user search through their files for a profile image. I encourage you to try and implement some of these ideas or even your own ideas.

In the following chapter, we will learn about layout management in PyQt.

CHAPTER 4

Learning About Layout Management

The previous chapters laid the foundation for getting started in PyQt5 as you learned to create GUIs with more functionality by adding widgets such as QPushButton and QCheckBox. Rather than continue to barrage you with PyQt's numerous widgets, taking a moment to learn about the various layout managers will save us some trouble moving forward.

Layout management is the way in which we arrange widgets in our windows. This can involve a number of different factors, including size and position, resize handling, and adding or removing widgets. Layout management is also very important to consider for the user looking at your application. Do a quick image search on the Internet for "worst GUI layouts" and you will see numerous applications crammed with widgets with no clear reasoning.

A **layout manager** is a class that contains methods which we can use to arrange widgets inside windows. They are useful for communicating between child and parent widgets to make them utilize the space in a window more efficiently.

In this chapter we are going to take a look at four methods that can be used for layout management in PyQt:

1. Absolute positioning with `move()`.

2. `QBoxLayout` which is useful for creating simple GUIs with horizontal or vertical layouts.

3. `QFormLayout` is a convenience layout useful for making application forms.

4. `QGridLayout` allows for more control over arranging widgets by specifying x and y coordinate values.

This chapter will also cover the idea of nesting layouts for creating more elaborate applications.

We will also take a look at a few new widgets and classes:

- QTextEdit – Similar to QLineEdit but creates a text entry field with more space

- QFileDialog – Native file dialog of PyQt that allows the user to open or save files

- QButtonGroup – To organize push button and checkbox widgets

- QSpinBox – A text box that displays integer values that the user can cycle through

- QComboBox – Presents a list of options in a dropdown-style widget

To make things simpler to understand in this chapter, let's first take a look at a few key concepts for using layout managers.

Choosing a Layout Manager

Setting the layout manager or changing the one we want to use in our applications isn't very difficult to do.

When you import modules in the beginning of your program, be sure to also include the layout manager(s) you want to use like so:

```
from PyQt5.QtWidgets import (QApplication, QWidget, QHBoxLayout,
QFormLayout)
```

Here we import both the QHBoxLayout and QFormLayout classes from QtWidgets. Then to set a specific layout manager inside a parent window or widget, we first must create an instance of that layout manager. In the following code, we call QVBoxLayout and then set the layout within the parent window using setLayout().

```
v_box = QVBoxLayout()
parent_window.setLayout(v_box)
```

Using a layout manager isn't necessary, but it is definitely preferred in order to make your applications easier to organize and to rearrange if necessary.

Customizing the Layout

So you've got your layout manager chosen. You've created your widgets. How do you go about adding them to the window?

```
name = QTextEdit()
v_box.addWidget(name)
```

In this code, we create a QTextEdit widget and then call the `addWidget()` method to add it in the layout. Be sure to consider what type of layout manager the parent window or widget is using before adding a child widget. For each layout manager, some of the parameters may change or you may need to call a different method to add widgets, but the concept is the same – create a widget, then add that widget into your layout. We will go over more details when we get to each specific manager.

If you need to create a more complex application with widgets arranged horizontally, vertically, or maybe even arranged in a grid, it is also possible to nest layouts in PyQt. **Nesting layouts** involves placing one layout manager inside of another. This can be accomplished by calling the `addLayout()` method and passing the name of a layout as a parameter.

One of the great things about using a layout manager is that when you resize the windows, the widgets in the window will all adjust accordingly. However, each layout manager has its own way of determining the spacing, alignment, size, or border around the widgets. These can all be manipulated and we will look at a few of these methods in the upcoming projects.

Absolute Positioning – Move()

While many people will recommend using layout managers, you can of course create layouts without them. This idea is called **absolute positioning** and it involves specifying the size and position values for each widget. This is the method we used in the previous chapters when we used **move()** to arrange widgets.

If you do decide to use absolute positioning, there are a few drawbacks to keep in mind. First of all, resizing the main window will not cause the widgets in it to adjust their size or position. Something else to keep in mind is the differences between operating systems, such as fonts and font sizes which could drastically change the look and layout of our GUI.

Absolute positioning can be most useful for setting the position and size values of widgets that are contained within other widgets.

Project 4.1 – Basic Notepad GUI

For our first project, let's take a look at creating a simple interface, a notepad GUI, to demonstrate how to use absolute positioning. A notepad is a way to capture our ideas or to take notes. It generally starts off blank and we fill in the information line by line. The benefit of having a digital notepad is that we can input and edit text much more easily than with real paper. Electronic notepads don't just include a blank area to write, but also tools which can be found at the top of the GUI window to help open, save, or edit notes.

This project, as can be seen in Figure 4-1, lays the foundation for our notepad GUI. In Chapter 5, we will take a look at how to improve upon this example by creating a menu interface and adding editing tools.

Figure 4-1. *Basic notepad GUI*

The QTextEdit Widget

If we are going to create a notepad GUI, then we need a text entry field that will allow us to enter and edit more than one line of text at a time.

The **QTextEdit** widget allows a user to enter text, either plain or rich text, and permits editing such as copy, paste, or cut. The widget can handle characters or paragraphs of text. Paragraphs are simply long strings that are word-wrapped into the widget and end with a newline character. QTextEdit is also useful for displaying lists, images, and tables or providing an interface for displaying text using HTML.

Take a look at the Solution code to the notepad GUI to see how to create a QTextEdit widget.

The QFileDialog Class

The **QFileDialog** class can be used to select files or directories found on your computer. This can be useful for locating and opening a file or looking for a directory to save a file and giving your file a name.

To open a file, we call the getOpenFileName() method, set the parent, create a title for the dialog box, display contents of a specific directory, and display files matching the patterns given in the string "All Files (*);;Text Files (*.txt)". You can also display image or other file types.

```
file_name = QFileDialog.getOpenFileName(self, 'Open File', "/Users/user_
name/Desktop/","All Files (*);;Text Files (*.txt)")
```

Saving a file is done in a similar fashion.

```
file_name = QFileDialog.getSaveFileName(self, 'Save File', "/Users/user_
name/Desktop/","All Files (*);;Text Files (*.txt)")
```

The look of the dialog box that appears will also reflect the type of system you are using. To change these properties, you could access QFileDialog.Options() and alter the dialog properties and appearance.

```
options = QFileDialog.Options()
options = QFileDialog.DontUseNativeDialog # By default native dialog is used
```

Basic Notepad GUI Solution

For this project, the GUI will consist of three widgets, two QPushButtons and a QTextEdit field (Listing 4-1). Users will be able to select the new button to clear the text in the line edit field or save the text to a file by clicking the save button and opening a dialog box.

Listing 4-1. Code for creating notepad GUI

```python
# notepad.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QPushButton, QTextEdit,
QMessageBox, QFileDialog)

class Notepad(QWidget):

    def __init__(self): # constructor
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 300, 400)
        self.setWindowTitle('4.1 - Notepad GUI')
        self.notepadWidgets()

        self.show()

    def notepadWidgets(self):
        """
        Create widgets for notepad GUI and arrange them in window
        """
        # Create push buttons for editing menu
        new_button = QPushButton("New", self)
        new_button.move(10, 20)
```

```python
        new_button.clicked.connect(self.clearText)

        save_button = QPushButton("Save", self)
        save_button.move(80, 20)
        save_button.clicked.connect(self.saveText)

        # Create text edit field
        self.text_field = QTextEdit(self)
        self.text_field.resize(280, 330)
        self.text_field.move(10, 60)

    def clearText(self):
        """
        If the new button is clicked, display dialog asking user if they
        want to clear the text edit field or not.
        """
        answer = QMessageBox.question(self, "Clear Text",
            "Do you want to clear the text?", QMessageBox.No | QMessageBox.Yes,
            QMessageBox.Yes)
        if answer == QMessageBox.Yes:
            self.text_field.clear()
        else:
            pass

    def saveText(self):
        """
        If the save button is clicked, display dialog to save the text in
        the text edit field to a text file.
        """
        options = QFileDialog.Options()
        notepad_text = self.text_field.toPlainText()

        file_name, _ = QFileDialog.getSaveFileName(self, 'Save File',
        "","All Files (*);;Text Files (*.txt)", options=options)

        if file_name:
            with open(file_name, 'w') as f:
                f.write(notepad_text)
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = Notepad()
    sys.exit(app.exec_())
```

Our notepad can be seen in Figure 4-1. The text displayed in the widgets shows that the QLineEdit widget can support different fonts, colors, and text sizes.

Explanation

When you use absolute positioning, you can think of the window as a grid where the top-left corner has the x and y coordinates (0, 0). If you create a window with height equal to 100 and width also 100, then the bottom-right corner has values (99, 99). To arrange widgets using move(), you need to specify values within the height and width range.

For example, the new_button push button is created and positioned as follows:

```
new_button = QPushButton("New", self)
new_button.move(10, 20) # x = 10, y = 20
```

Three widgets are created in Listing 4-1, two buttons and a QTextEdit widget for inputting text. We can use the buttons to either clear text or save the text we have typed.

When the user clicks the save_button, the saveText() method is called which displays a QFileDialog like the one shown in Figure 4-2.
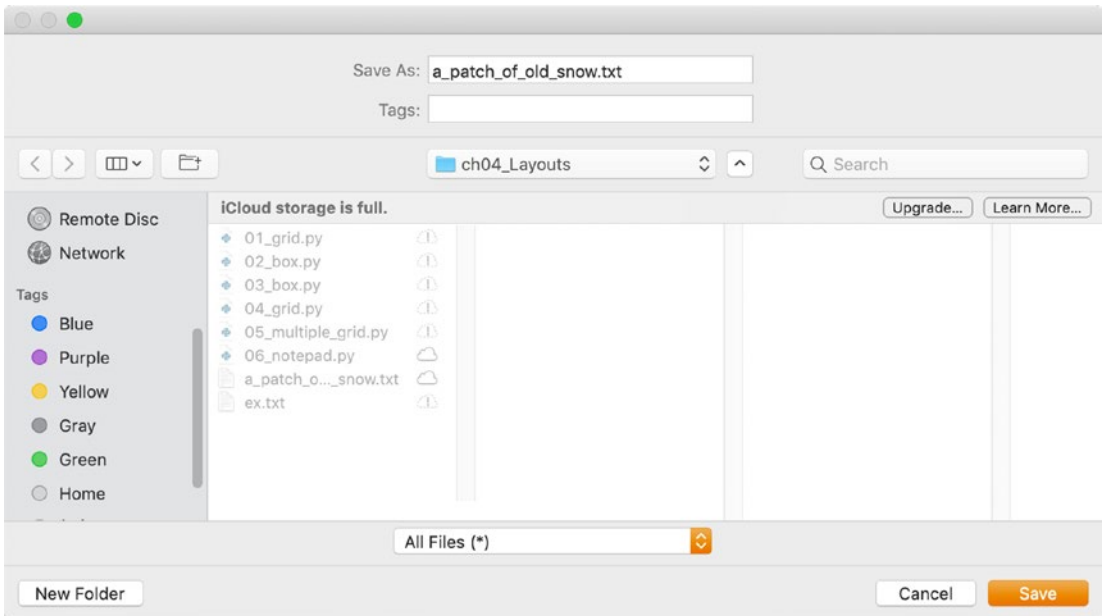
Figure 4-2. *QFileDialog box that opens to save the text from the notepad GUI*

The contents of the text file are shown in Figure 4-3. Because we save the file as a text file, it loses some information but still retains the spacing and paragraphs separated by newlines. To keep the rich text information, you could save the file using HTML. This will be covered in Chapter 5.

Figure 4-3. *Text file showing the text saved from notepad GUI*

61

The QHBoxLayout and QVBoxLayout Classes

Arranging widgets can be accomplished easily with the QBoxLayout classes. PyQt has two different QBoxLayout styles, QHBoxLayout and QVBoxLayout:

- **QHBoxLayout** – Used to arrange widgets horizontally from left to right in the window

- **QVBoxLayout** – Used to arrange widgets vertically from top to bottom in the window

Creating a basic GUI with only one of these layout managers is possible, but the real potential comes from being able to combine the two of them to create more elaborate layouts. Together we can combine them, along with the `addStretch()` method, to place widgets anywhere in the window. Think of `addStretch()` as an adjustable blank space that can be used to help arrange widgets relative to each other or to help place widgets in the window.

In the following project, we will take a look at how to use QHBoxLayout and QVBoxLayout in the same program to create a simple survey GUI application.

Project 4.2 – Survey GUI

Creating a survey to collect data from users can be very useful for businesses or for research. In the following project, we will take a look at how to use the QBoxLayout class to create a simple window that displays a question to the user and allows them to select an answer.

From personal experience, the Python language is very good at automating repetitive tasks. In university, I needed to collect data from almost a thousand participants in order to research marketing trends related to how they spent money at sporting events. For my research I decided to create an application that would ask the user a question and then store their answers in a Python list. When a participant reached the end of the survey, their answers were written to a file. This greatly helped later when I needed to use that same data to create graphs and charts.

Figure 4-4 shows the program we are going to make in Project 4.2.
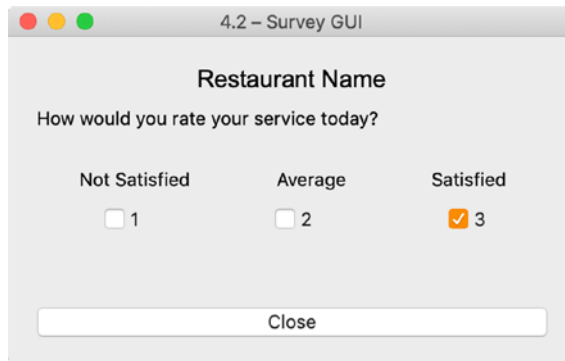
Figure 4-4. *Survey GUI*

The QButtonGroup Class

You may often have a few checkboxes or buttons that need to be grouped together to make it easier to manage them. Luckily, PyQt has the **QButtonGroup** class to help not only group and arrange buttons together, but also has the ability to make buttons mutually exclusive. This is also helpful if you only want one checkbox to be checked at a time.

QButtonGroup is not actually a widget, but a container where you can add widgets. Therefore, you can't actually add QButtonGroup to a layout. The following code shows the method of how to import and set up QButtonGroup in your application:

```
from PyQt5.QtWidgets import QButtonGroup, QCheckBox

b_group = QButtonGroup() # Create instance of QButtonGroup
# Create two checkboxes
cb_1 = QCheckBox("CB 1")
cb_2 = QCheckBox("CB 2")

# Add checkboxes into QButtonGroup
b_group.addButton(cb_1)
b_group.addButton(cb_2)
# Connect all buttons in a group to one signal
b_group.buttonClicked.connect(cbClicked)

def cbClicked(cb):
print(cb)
```

In the above code we create two QCheckBox widgets and add them to the QButtonGroup using the `addButton()` method. To make the buttons mutually exclusive, we check to see if a signal is sent not from each individual button but from the button group instead. This is done with

```
b_group.buttonClicked.connect(cbClicked)
```

Survey GUI Solution

The survey GUI consists of QLabel widgets to display the title, question, and ratings labels for each checkbox. For the checkboxes in the window, the text beside each label could have been left blank, but the numbers are left as a visual cue to the user. Once the user selects a choice, they can close the window using a QPushButton (Listing 4-2).

Listing 4-2. Code for creating survey GUI

```python
# survey.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QPushButton,
QCheckBox, QButtonGroup, QHBoxLayout, QVBoxLayout)
from PyQt5.QtGui import QFont

class DisplaySurvey(QWidget):
    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 400, 230)
        self.setWindowTitle('4.2 - Survey GUI')
        self.displayWidgets()

        self.show()
```

```python
def displayWidgets(self):
    """
    Set up widgets using QHBoxLayout and QVBoxLayout.
    """
    # Create label and button widgets
    title = QLabel("Restaurant Name")
    title.setFont(QFont('Arial', 17))
    question = QLabel("How would you rate your service today?")

    # Create horizontal layouts
    title_h_box = QHBoxLayout()
    title_h_box.addStretch()
    title_h_box.addWidget(title)
    title_h_box.addStretch()

    ratings = ["Not Satisfied", "Average", "Satisfied"]

    # Create checkboxes and add them to horizontal layout, and add
      stretchable
    # space on both sides of the widgets
    ratings_h_box = QHBoxLayout()
    ratings_h_box.setSpacing(60) # Set spacing between in widgets in
    horizontal layout

    ratings_h_box.addStretch()
    for rating in ratings:
        rate_label = QLabel(rating, self)
        ratings_h_box.addWidget(rate_label)
    ratings_h_box.addStretch()

    cb_h_box = QHBoxLayout()
    cb_h_box.setSpacing(100) # Set spacing between in widgets in
    horizontal layout
    # Create button group to contain checkboxes
    scale_bg = QButtonGroup(self)

    cb_h_box.addStretch()
    for cb in range(len(ratings)):
```

```python
            scale_cb = QCheckBox(str(cb), self)
            cb_h_box.addWidget(scale_cb)
            scale_bg.addButton(scale_cb)
        cb_h_box.addStretch()

        # Check for signal when checkbox is clicked
        scale_bg.buttonClicked.connect(self.checkboxClicked)

        close_button = QPushButton("Close", self)
        close_button.clicked.connect(self.close)

        # Create vertical layout and add widgets and h_box layouts
        v_box = QVBoxLayout()
        v_box.addLayout(title_h_box)
        v_box.addWidget(question)
        v_box.addStretch(1)
        v_box.addLayout(ratings_h_box)
        v_box.addLayout(cb_h_box)
        v_box.addStretch(2)
        v_box.addWidget(close_button)

        # Set main layout of the window
        self.setLayout(v_box)

    def checkboxClicked(self, cb):
        """
        Print the text of checkbox selected.
        """
        print("{} Selected.".format(cb.text()))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = DisplaySurvey()
    sys.exit(app.exec_())
```

The survey GUI can be seen in Figure 4-4.

Explanation

After importing all of the PyQt classes and setting up the `DisplaySurvey` class, we begin by creating some labels, setting up the text for the ratings labels, and creating the `close_button`.

The application consists of three separate QHBoxLayout objects – `title_h_box`, `ratings_h_box`, and `cb_h_box` – and a single QVBoxLayout layout, `v_box`. For this GUI, `v_box` will act as the container for all of the other widgets and layouts, arranged vertically from top to bottom.

Combining Box Layouts and Arranging Widgets

When we say combining layouts, what that really means is nesting one type of box layout inside of another type to get the benefit of vertical or horizontal layouts.

The following bit of code shows how to create a QHBoxLayout object and add a widget to it:

```
# Create horizontal layouts
title_h_box = QHBoxLayout()
title_h_box.addStretch()
title_h_box.addWidget(title)
title_h_box.addStretch()
```

The `addStretch()` method acts like an invisible widget that can be used to help arrange widgets in a layout manager. Widgets in QHBoxLayout are organized left to right, so in `title_h_box`, `addStretch` is added to the left, `title` in the middle, and another `addStretch` to the right. This centers the `title` in `title_h_box`.

To add the rating labels and checkboxes to the window, a separate QHBoxLayout is created for each one. Each widget added is spaced out using the `setSpacing()` method, which is useful for creating a fixed amount of space between widgets inside of a layout.

Adding layouts or widgets to a parent layout is as simple as changing the method called.

```
v_box = QVBoxLayout() # Create vertical layout
v_box.addLayout(title_h_box) # Add horizontal layout
v_box.addWidget(question) # Add widget
```

The QFormLayout Class

In Chapter 3 we looked at how to make a create new user GUI (Project 3.2) that would collect a user's information. In that project, each line consisted of a QLabel widget on the left and a QLineEdit widget on the right. They were then arranged in the window using absolute positioning.

For situations like this where you need to create a form to collect information from a user, PyQt provides the **QFormLayout** class. It is a layout class that arranges its children widgets into a two-column layout, the left column consisting of labels and the right one consisting of entry field widgets such as QLineEdit or QSpinBox. The QFormLayout class makes designing these kinds of GUIs very convenient.

Project 4.3 – Application Form GUI

We all have to fill out application forms at some point, applying for a job, when you want to go to university, trying to get insurance for your car, or signing up for a new bank account.

For this project let's take a look at creating an application form that someone could use to set up an appointment at the hospital like in Figure 4-5. When filling out an electronic application, you can combine many different widgets to not only reduce the size of the window but also minimize the amount of clutter from text that you might usually see on a paper application.

Before getting started on the application form, we should learn about two new widgets – QSpinBox and QComboBox – that we will use in the application GUI.

Figure 4-5. *Application form GUI*

The QSpinBox and QComboBox Widgets

Rather than using a QLineEdit widget for the user to input information, sometimes you may want them to only be allowed to select from a list of predetermined values. Both QSpinBox and QComboBox widgets are very useful for handling this kind of task.

QSpinBox creates an object that is similar to a text box, but allows the user to select integer values either by typing a value into the widget or by clicking the up and down arrows. You can also edit the range of the values, set the step size when the arrow is clicked, set a starting value, or even add prefixes or suffixes in the box. There are also other kinds of spin boxes in PyQt, such as QDateEdit, to select date and time values.

QComboBox is a way to display a list of options for the user to select from. When a user clicks the arrow button, a pop-up list appears and displays a collection of possible selections.

In Listing 4-3 we will take a look at how to create both kinds of objects, add them to our layout, and find out how to use the values in QSpinBox to update other widgets in the GUI window.

Listing 4-3. Code for creating QSpinBox and QComboBox widgets

```python
# spin_combo_boxes.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QComboBox,
QSpinBox, QHBoxLayout, QVBoxLayout)
from PyQt5.QtGui import QFont
from PyQt5.QtCore import Qt

class SelectItems(QWidget):
    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 300, 200)
        self.setWindowTitle('ComboBox and SpinBox')
        self.itemsAndPrices()

        self.show()

    def itemsAndPrices(self):
        """
        Create the widgets so users can select an item from the combo boxes
        and a price from the spin boxes
        """
        info_label = QLabel("Select 2 items you had for lunch and their prices.")
        info_label.setFont(QFont('Arial', 16))
        info_label.setAlignment(Qt.AlignCenter)
        self.display_total_label = QLabel("Total Spent: $")
        self.display_total_label.setFont(QFont('Arial', 16))
        self.display_total_label.setAlignment(Qt.AlignRight)
```

```python
# Create list of food items and add those items to two separate
  combo boxes
lunch_list = ["egg", "turkey sandwich", "ham sandwich", "cheese",
"hummus", "yogurt", "apple", "banana", "orange", "waffle", "baby
carrots", "bread", "pasta", "crackers", "pretzels", "pita chips",
"coffee", "soda", "water"]

lunch_cb1 = QComboBox()
lunch_cb1.addItems(lunch_list)
lunch_cb2 = QComboBox()
lunch_cb2.addItems(lunch_list)

# Create two separate price spin boxes
self.price_sb1 = QSpinBox()
self.price_sb1.setRange(0,100)
self.price_sb1.setPrefix("$")
self.price_sb1.valueChanged.connect(self.calculateTotal)

self.price_sb2 = QSpinBox()
self.price_sb2.setRange(0,100)
self.price_sb2.setPrefix("$")
self.price_sb2.valueChanged.connect(self.calculateTotal)

# Create horizontal boxes to hold combo boxes and spin boxes
h_box1 = QHBoxLayout()
h_box2 = QHBoxLayout()

h_box1.addWidget(lunch_cb1)
h_box1.addWidget(self.price_sb1)
h_box2.addWidget(lunch_cb2)
h_box2.addWidget(self.price_sb2)

# Add widgets and layouts to QVBoxLayout
v_box = QVBoxLayout()
v_box.addWidget(info_label)
v_box.addLayout(h_box1)
v_box.addLayout(h_box2)
v_box.addWidget(self.display_total_label)
```

```
        self.setLayout(v_box)

    def calculateTotal(self):
        """
        Calculate and display total price from spin boxes and change value
        shown in QLabel
        """
        total = self.price_sb1.value() + self.price_sb2.value()
        self.display_total_label.setText("Total Spent: ${}".
        format(str(total)))

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = SelectItems()
    sys.exit(app.exec_())
```

Your window should look similar to the one seen in Figure 4-6.
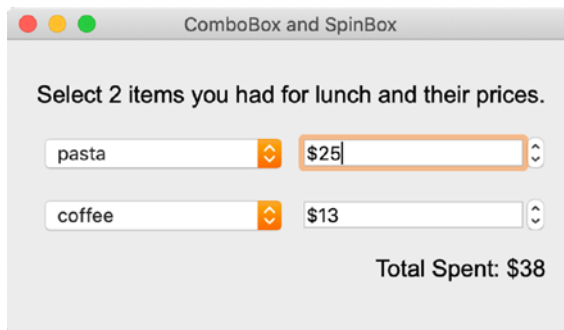
Figure 4-6.   *GUI to show how to create QSpinBox and QComboBox widgets*

Explanation

The code for this application is also another demonstration of how to create nested layouts in PyQt. Here we create two instances of QHBoxLayout and add them to a vertical layout.

We create two separate combo boxes, `lunch_cb1` and `lunch_cb2`, and add the list of items that we want to be displayed to each of them using the `addItems()` method. Then two separate spin boxes are created, `price_sb1` and `price_sb2`.

```
self.price_sb1.setRange(0,100)
self.price_sb1.setPrefix("$")
```

The `setRange()` method is used to set the upper and lower boundaries for a spin box and `setPrefix()` can be used to display other text inside of the text box, in this case a dollar sign. This can be helpful to give the user more information about the widget's purpose. All of these widgets are then added to two separate horizontal layouts, `h_box1` and `h_box2`.

Note   Since the two QComboBox objects and the two QSpinBox objects each contain the same values, you may have the urge to just try and use them over again when adding them to QVBoxLayout. This won't work. When you add an item to a widget or to a layout, that widget takes ownership of the item. This means you cannot add an item to more than one widget or layout. You will need to create a new instance.

Finally, as we change the values in the spin boxes, they both send a signal that is connected to the `calculateTotal()` method. This will dynamically update the value for `display_total_label` in the window.

Application Form GUI Solution

The application form GUI consists of a number of different widgets, including QLabel, QLineEdit, QSpinBox, QComboBox, QTextEdit, and QPushButton (Listing 4-4). Nesting layouts is also possible with the QFormLayout manager.

Listing 4-4. Code for creating application form GUI

```
# application.py
# Import necessary modules
import sys
```

```python
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QPushButton,
QFormLayout, QLineEdit, QTextEdit, QSpinBox, QComboBox, QHBoxLayout)
from PyQt5.QtGui import QFont
from PyQt5.QtCore import Qt

class GetApptForm(QWidget):
    def __init__(self):
        super().__init__()

        self.initializeUI()


    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 300, 400)
        self.setWindowTitle('4.3 - Application Form GUI')
        self.formWidgets()

        self.show()

    def formWidgets(self):
        """
        Create widgets that will be used in the application form.
        """
        # Create widgets
        title = QLabel("Appointment Submission Form")
        title.setFont(QFont('Arial', 18))
        title.setAlignment(Qt.AlignCenter)

        name = QLineEdit()
        name.resize(100, 100)
        address = QLineEdit()
        mobile_num = QLineEdit()
        mobile_num.setInputMask("000-000-0000;")

        age_label = QLabel("Age")
        age = QSpinBox()
        age.setRange(1, 110)
```

```python
height_label = QLabel("Height")
height = QLineEdit()
height.setPlaceholderText("cm")
weight_label = QLabel("Weight")
weight = QLineEdit()
weight.setPlaceholderText("kg")

gender = QComboBox()
gender.addItems(["Male", "Female"])

surgery = QTextEdit()
surgery.setPlaceholderText("separate by ','")
blood_type = QComboBox()
blood_type.addItems(["A", "B", "AB", "O"])

hours = QSpinBox()
hours.setRange(1, 12)
minutes = QComboBox()
minutes.addItems([":00", ":15", ":30", ":45"])
am_pm = QComboBox()
am_pm.addItems(["AM", "PM"])

submit_button = QPushButton("Submit Appointment")
submit_button.clicked.connect(self.close)

# Create horizontal layout and add age, height, and weight to h_box
h_box = QHBoxLayout()
h_box.addSpacing(10)
h_box.addWidget(age_label)
h_box.addWidget(age)
h_box.addWidget(height_label)
h_box.addWidget(height)
h_box.addWidget(weight_label)
h_box.addWidget(weight)

# Create horizontal layout and add time information
desired_time_h_box = QHBoxLayout()
desired_time_h_box.addSpacing(10)
```

```
        desired_time_h_box.addWidget(hours)
        desired_time_h_box.addWidget(minutes)
        desired_time_h_box.addWidget(am_pm)

        # Create form layout
        app_form_layout = QFormLayout()

        # Add all widgets to form layout
        app_form_layout.addRow(title)
        app_form_layout.addRow("Full Name", name)
        app_form_layout.addRow("Address", address)
        app_form_layout.addRow("Mobile Number", mobile_num)
        app_form_layout.addRow(h_box)
        app_form_layout.addRow("Gender", gender)
        app_form_layout.addRow("Past Surgeries ", surgery)
        app_form_layout.addRow("Blood Type", blood_type)
        app_form_layout.addRow("Desired Time", desired_time_h_box)
        app_form_layout.addRow(submit_button)

        self.setLayout(app_form_layout)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = GetApptForm()
    sys.exit(app.exec_())
```

When completed, your GUI should look similar to Figure 4-5.

Note   Depending upon what system you are working on, the look and layout of the widgets in your window will change.

Explanation

Using the QFormLayout class is pretty straightforward. In the formWidgets() method, the widgets that will be used in this GUI are instantiated in the beginning. An important one to point out is the mobile_num line edit object.

Any type of character can naturally be typed into the QLineEdit entry field. However, if you want to limit the type, size, or manner in which characters can be input, then you can create an input mask by calling the setInputMask() method. setInputMask() also can be used to set the maximum number of characters.

Two parts of this application have widgets arranged horizontally. For these widgets we will add them in QHBoxLayout objects.

The QFormLayout object is created by

```
app_form_layout = QFormLayout()
```

Next, all widgets and layouts are added to the form layout using the addRow() method. Finally, the layout for our window is set using self.setLayout(app_form_layout).

Widgets and layouts can be added to a QFormLayout object in the following ways:

```
form_layout.addRow(QWidget)
form_layout.addRow("text", QWidget)
form_layout.addRow(layout)
```

The first one will add a widget and may cause it to stretch to fit the window. The second fits the text and its widget into a two-column layout. The last one can be used to nest layouts.

The QGridLayout Class

The **QGridLayout** layout manager is used to arrange widgets in rows and columns similar to a spreadsheet or matrix. The layout manager takes the space within its parent window or widget and divides it up according to the sizes of the widgets within that row (or column). Adding space between widgets, creating a border, or stretching widgets across multiple rows or columns is also possible.

Understanding how to add and manipulate widgets using QGridLayout is also easier. The grid for the layout manager starts at value (0, 0) which is the top leftmost cell. To add a widget underneath it (the next row), simply add 1 to the left value, (1, 0). To keep moving down the rows, keep increasing the left value. To move across columns, increase the right value.

Let's take a look at how to make a to-do list using QGridLayout.

Project 4.4 – To-Do List GUI

We all have things that we must do every day, and many of us need a way to help organize our busy lives. For this project we will take a look at creating a basic layout for a to-do list.

Some to-do lists are broken down by hours of the day, by importance of goals, or by various other tasks we may need to do for that day, week, or even month. Once a goal is complete, we need some way to check off a task or remove it.

The project will consist of a to-do list made up of two parts, a list of things you must do on the left and daily appointments on the right. The "Must Dos" will consist of QCheckBox and QLineEdit widgets. The "Appointments" will be separated into three sections, morning, noon, and evening, and will use QTextEdit widgets to give the user an area to write down their tasks. You can see the GUI we will be building in Figure 4-7.
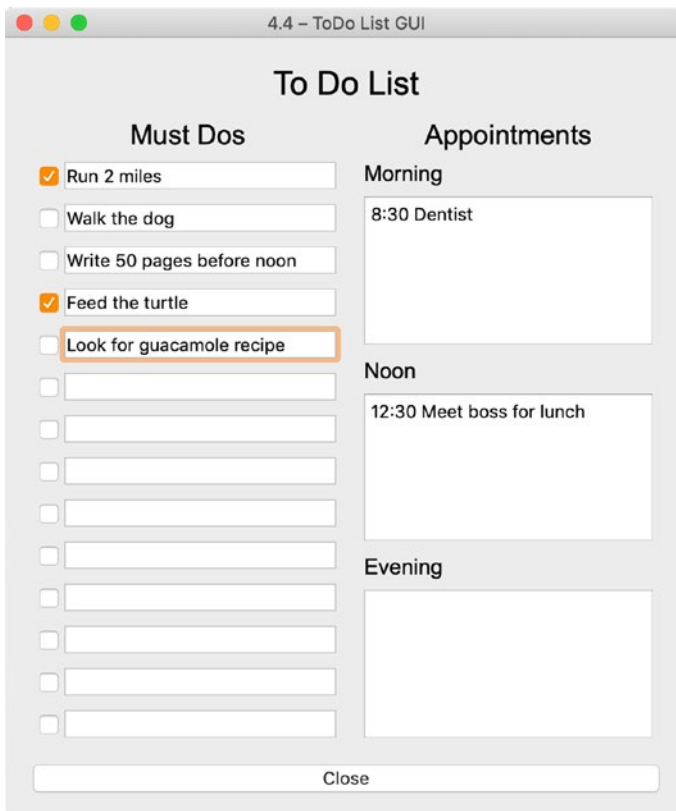
Figure 4-7. *To-do list GUI*

To-Do List GUI Solution

For this project (Listing 4-5), we will be focusing mainly on how to create the GUI and arrange widgets using the QGridLayout class as the main layout. This project also includes a nested QVBoxLayout for the "Appointments" layout.

Listing 4-5. Code for the to-do list GUI

```python
# todolist.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QTextEdit,
QLineEdit, QPushButton, QCheckBox, QGridLayout, QVBoxLayout)
from PyQt5.QtGui import QFont
from PyQt5.QtCore import Qt

class ToDoList(QWidget):

    def __init__(self): # Constructor
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 500, 350)
        self.setWindowTitle('4.4 – ToDo List GUI')
        self.setupWidgets()

        self.show()

    def setupWidgets(self):
        """
        Create widgets for to-do list GUI and arrange them in the window
        """
        # Create grid layout
```

```python
main_grid = QGridLayout()

todo_title = QLabel("To Do List")
todo_title.setFont(QFont('Arial', 24))
todo_title.setAlignment(Qt.AlignCenter)

close_button = QPushButton("Close")
close_button.clicked.connect(self.close)

# Create section labels for to-do list
mustdo_label = QLabel("Must Dos")
mustdo_label.setFont(QFont('Arial', 20))
mustdo_label.setAlignment(Qt.AlignCenter)
appts_label = QLabel("Appointments")
appts_label.setFont(QFont('Arial', 20))
appts_label.setAlignment(Qt.AlignCenter)

# Create must-do section
mustdo_grid = QGridLayout()
mustdo_grid.setContentsMargins(5, 5, 5, 5)

mustdo_grid.addWidget(mustdo_label, 0, 0, 1, 2)

# Create checkboxes and line edit widgets
for position in range(1, 15):
    checkbox = QCheckBox()
    checkbox.setChecked(False)
    linedit = QLineEdit()
    linedit.setMinimumWidth(200)
    mustdo_grid.addWidget(checkbox, position, 0)
    mustdo_grid.addWidget(linedit, position, 1)

# Create labels for appointments section
morning_label = QLabel("Morning")
morning_label.setFont(QFont('Arial', 16))
morning_entry = QTextEdit()
noon_label = QLabel("Noon")
noon_label.setFont(QFont('Arial', 16))
```

```
        noon_entry = QTextEdit()
        evening_label = QLabel("Evening")
        evening_label.setFont(QFont('Arial', 16))
        evening_entry = QTextEdit()

        # Create vertical layout and add widgets
        appt_v_box = QVBoxLayout()
        appt_v_box.setContentsMargins(5, 5, 5, 5)

        appt_v_box.addWidget(appts_label)
        appt_v_box.addWidget(morning_label)
        appt_v_box.addWidget(morning_entry)
        appt_v_box.addWidget(noon_label)
        appt_v_box.addWidget(noon_entry)
        appt_v_box.addWidget(evening_label)
        appt_v_box.addWidget(evening_entry)

        # Add other layouts to main grid layout
        main_grid.addWidget(todo_title, 0, 0, 1, 2)
        main_grid.addLayout(mustdo_grid, 1, 0)
        main_grid.addLayout(appt_v_box, 1, 1)
        main_grid.addWidget(close_button, 2, 0, 1, 2)

        self.setLayout(main_grid)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = ToDoList()
    sys.exit(app.exec_())
```

When you are finished, your GUI should look similar to Figure 4-7.

Explanation

Here we start by creating our ToDoList class which inherits from QWidget. In the setupWidgets() method, a few QLabel widgets which will serve as header labels for the GUI and the different sections are created. A QPushButton, which will close the program, is also instantiated.

The must-do section uses the QGridLayout class. A margin can be set to frame a layout using the setContentMargins() method.

```
mustdo_grid.setContentsMargins(5, 5, 5, 5)
```

Each integer specifies the size of the border in pixels, (left, top, right, bottom).

Then the layout managers, mustdo_grid and appt_v_box, and their widgets are created and added. Finally, the title, the two layouts, and the close button are added to the main_grid layout, and main_grid is set as the main layout using setLayout().

Adding Widgets and Spanning Rows and Columns with QGridLayout

Since widgets will be placed in a grid-like structure, when you add a new object to the layout, you must specify the row and column values as parameters of the addWidget() or addLayout() methods. Take a look at the following lines:

```
main_grid.addWidget(todo_title, 0, 0)
main_grid.addLayout(mustdo_grid, 1, 0)
main_grid.addLayout(appt_v_box, 1, 1)
```

The todo_title QLabel widget is added to the main_grid layout at the position where the row equals 0 and column equals 0, which is also the top-left corner. Then, the mustdo_grid is added directly below it by increasing the row value to 1 and leaving the column value equal to 0. Finally, we move over one column for the appt_v_box layout by setting the column value to 1. If you want to build a GUI with more widgets using QGridLayout, then you would just continue in this manner moving away from 0, 0.

But what happens if you have a widget in a column or a row that is next to another widget that needs to take up more space in the vertical or horizontal direction? QGridLayout allows us to specify the number of rows or columns that we want a single widget or layout to **span**. Spanning can be thought of as stretching a widget horizontally or vertically to help us better arrange our GUI.

```
main_grid.addWidget(clear_button, 2, 0, 1, 3)
```

The extra two parameters at the end, 1 and 3, tell the layout manager that we want to span one row and three columns. This causes the widget to stretch horizontally.

Summary

Taking the time to learn about layout management will save you time and effort when coding your own GUI applications. In this chapter we reviewed absolute positioning using the move() method and learned about three of PyQt's layout managers – QBoxLayout, QFormLayout, and QGridLayout. Each of these classes has their own special purpose, but one of the real powers of PyQt is how convenient it is to nest them into other layouts to make more complex GUIs.

It is important to note that any of the subclasses within QWidget can also use a layout manager to manage their children. The advantages of using a layout manager include

- Positioning of child widgets

- Setting default sizes for windows

- Handling resizing of windows

- Updating content in the window or parent widget when something changes, such as type of font, font size, and hiding, showing, or removing of a child widget

You can actually design and lay out your interface graphically using the Qt Designer Application. We will take a brief look at how to do this in Chapter 7.

In Chapter 5 we are going to take a look at how to add menus to our applications.

Menus, Toolbars, and More

As you add more and more features to your applications, you will need some way to present all of the individual options to the user. A **menu** is a list of commands that a computer program can perform presented in a more manageable and organized fashion. No matter what type of device or application you are using or what kind of menu system it has, if it has a menu in place, its role is to help you navigate through the various operations in order to help you select the tasks you wish to perform.

Graphical user interfaces have numerous kinds of menus, such as context menus or pull-down menus, and can contain a variety of text and symbols. These symbols can be selected to give the computer an instruction. Think about a text editing program and all the various icons at the top, for example, open a file, save a file, select font, or select color. All of these symbols and text represent some task that the application can perform presented to the user in a manner that should promote better ease of use.

In this chapter we are going to take a look at how to create menus and toolbars for GUI applications in PyQt5. Everything up until now has been used to build a foundation for creating user interfaces, from PyQt classes and widgets to layout design.

Different from previous chapters, here we will be looking at how to make completely functioning programs, a notepad GUI and a simple photo editor GUI. These applications can either be used right away or as a starting point for building your own GUI program. There is a fair amount of information, from new concepts to additional widgets and classes, covered in this chapter.

For menus using PyQt, you will take a look at

1. The `QMainWindow` class for setting up the main window

2. Creating `QMenubar` and `QMenu` objects

3. Adding actions to menus using the `QAction` class

4.   Setting up the status bar using `QStatusBar` to display information about actions

5.   Using `QDockWidget` to build detachable widgets to hold an application's tools

6.   How to create submenus and checkable menu items

Other concepts and widgets covered include

- Setting and changing icons in the main window and on widgets with `QIcon`

- New types of dialogs including `QInputDialog`, `QColorDialog`, `QFontDialog`, and `QMessageBox`'s `About` dialog box

- How to handle and manipulate images using `QPixmap` and `QTransform` classes

- How to print images using `QPrinter` and `QPainter`

Let's jump right into coding a basic menu framework that will help you learn about creating menus with PyQt5 and some new classes, QMainWindow and QMenu.

Create a Basic Menu

For this first part, we will be taking a look at how to create a simple menubar. A **menubar** is a set of pull-down menus with list commands that we can use to interact with the program. In this program, the menubar will contain one menu with only one command, Exit (Listing 5-1).

Listing 5-1. Basic structure for creating the menu in an application

```
# menu_framework.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QAction)

class BasicMenu(QMainWindow):
```

```python
    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 350, 350) # x, y, width, height
        self.setWindowTitle('Basic Menu Example')

        self.createMenu()

        self.show()

    def createMenu(self):
        """
        Create skeleton application with a menubar
        """
        # Create actions for file menu
        exit_act = QAction('Exit', self)
        exit_act.setShortcut('Ctrl+Q')
        exit_act.triggered.connect(self.close)

        # Create menubar
        menu_bar = self.menuBar()
        menu_bar.setNativeMenuBar(False)

        # Create file menu and add actions
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(exit_act)
# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = BasicMenu()
    sys.exit(app.exec_())
```

Figure 5-1 shows what adding a simple menu will look like on MacOS. Notice how in the left image File is displayed in the menubar, and when we scroll over it using our mouse, the Exit option is shown in the image on the right.
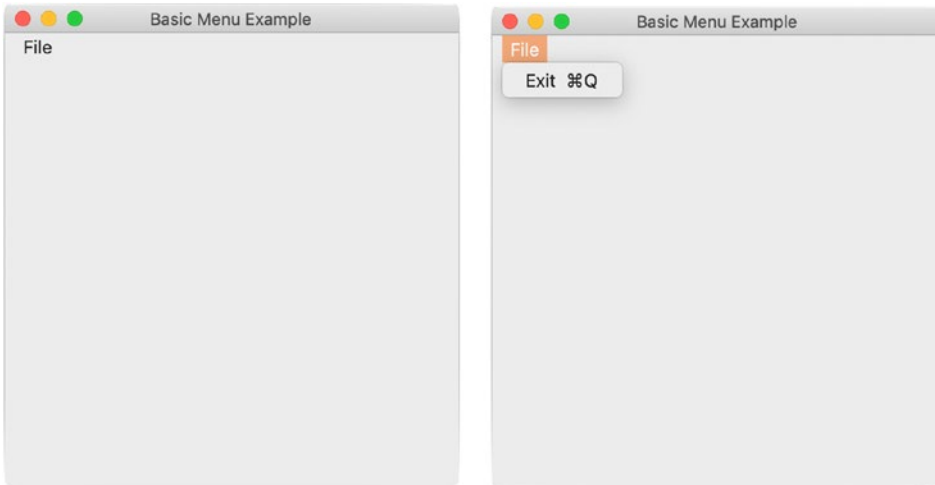
Figure 5-1. *A menubar is created (left) displaying our File menu. A pull-down menu is displayed (right) with one command, Exit*

Explanation

The framework for this program contains no widgets, but does show how to set up a simple File menu located in the top-left corner of the GUI. Take a look at the beginning of the program and notice the classes being imported from QtWidgets. We still import QApplication, but there are also two new classes, QMainWindow and QAction. You may also notice that this time there is no QWidget.

The QMainWindow class provides the necessary tools for building an application's graphical user interface. Notice that the BasicMenu class in the preceding code is written as

```
class BasicMenu(QMainWindow)
```

The class to build our window inherits from QMainWindow instead of QWidget.

QMainWindow vs. QWidget

The **QMainWindow** class focuses on creating and managing the layout for the main window of an application. It allows you to set up a window with a status bar, a toolbar, dock widgets, or other menu options in predefined places all designed around functions that the main window should have.

The QWidget class is the base class for all user interface objects in Qt. The widget is the basic building block of GUIs. It is interactive, allowing the user to communicate with the computer to perform some task. Many of the widgets you have already looked at, such as QPushButton and QTextEdit, are just subclasses of QWidget that give functionality to your programs.

A window in an application is really just a widget that is not embedded in a parent widget. What is important to understand is that QMainWindow actually inherits from the QWidget class. It is a special purpose class focusing mainly on creating menus and housing widgets in your program. In Figure 5-2, you can see how the different widgets that QMainWindow can use have areas specifically assigned for them. Take a look at the image to see how the menubar, dock widgets, and the central widget can be arranged inside of the main window.

The central widget in the center of the window must be set if you are going to use QMainWindow as your base class. For example, you could use a single QTextEdit widget or create a QWidget object to act as a parent to a number of other widgets, then use setCentralWidget(), and set your central widget for the main window.
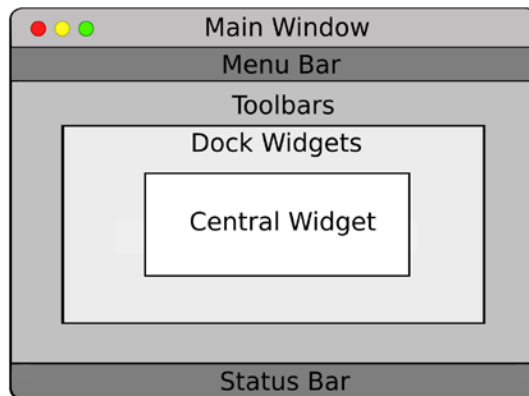
Figure 5-2. *Example layout for QMainWindow class (Adapted from* `https://doc.qt.io/` *web site)*

Creating the Menubar and Adding Actions

At the top of the window in Figure 5-1, you will see the menubar which contains one menu, File. In order to create a menubar, you must create an instance of the **QMenuBar** class, which we created by

```
menu_bar = self.menuBar()
```

You could create a menubar by actually calling the QMenuBar class, but it is just as easy to create a menubar using the menuBar() function provided by QMainWindow.

Note   Due to guidelines set by the MacOS system, you must set the property to use the platform's native settings to False. Otherwise, the menu will not appear in the window. You can do this with menu_bar.setNativeMenuBar(False). For those using Windows or Linux, you can comment this line out or delete it completely from your code.

Adding menus to the menubar is also really simple in PyQt:

```
file_menu = menu_bar.addMenu('File')
```

Here we are using the addMenu() method to add a menu named File to the menu_bar. Using addMenu() adds a **QMenu** object to our menubar. Once again, it is just as simple to use the functions provided by the QMainWindow class.

A menu contains a list of action items such as Open, Close, and Find. In PyQt, these actions are created from the **QAction** class, defining actions for menus and toolbars. Many actions in an application are also given shortcut keys making it easier to perform that task, for example, Ctrl+V for the paste action (Cmd+V on MacOS) or Ctrl+X for the cut action (Cmd+X on MacOS). Take a look at how the Exit action is created and then added to file_menu.

```
exit_act = QAction('Exit', self)
exit_act.setShortcut('Ctrl+Q')
exit_act.triggered.connect(self.close)
```

The Exit action, exit_act, is an instance of the QAction class. In the next line the shortcut for the exit_act is set explicitly using the setShortcut() method with the key combination Ctrl+Q. Another way to set the shortcut is to use the ampersand key, &, in front of the letter you want to use as the shortcut. For example,

```
open_act = QAction('&Open', self)
```

Note   By default, on MacOS shortcuts are disabled. The best way to use them is with setShortcut().

Similar to QPushButtons, actions in the menu emit a signal and need to be connected to a slot in order to perform an action. This is done using `triggered.connect()`. Using the QAction class is very useful since many common commands can be invoked through the menu, toolbars, or shortcuts and need to be able to perform correctly no matter which widget invokes the action.

Setting Icons with the QIcon Class

In GUI applications, icons are small graphical images or symbols that can represent an action the user can perform. They are often used to help the user more quickly locate common actions and better navigate an application. For example, in a word editing program such as Microsoft Word, the toolbar at the top of the GUI contains a large amount of icons, each with icon and textual descriptions.

Chapter 2 briefly introduced the QPixmap class which is used for handling images. The **QIcon** class provides methods that can use pixmaps and modify their style or size to be used in an application. One really great use of QIcon is to set the appearance of an icon representing an action to active or disabled.

Setting icons is very useful not only for the actions in a toolbar but also for setting the application icon that is displayed in the title bar of the GUI window. Actions can be in four states, represented by icons: Normal, Disabled, Active, or Selected. QIcon can also be used when setting the icons on other widgets, as well.

Listing 5-2 shows how to reset the application icon displayed in the main window and how to set the icon on a QPushButton.

Note   For MacOS users, the application window cannot be changed due to system guidelines. You should still take a look at this program though, as it also shows how to set icons for other widgets in PyQt.

Listing 5-2. Code to show how to set icons for the main window and on QPushButtons

```
# change_icons.py
# Import necessary modules
import sys
```

```python
from PyQt5.QtWidgets import (QApplication, QLabel, QWidget, QPushButton,
QVBoxLayout)
from PyQt5.QtGui import QIcon
from PyQt5.QtCore import QSize
import random

class ChangeIcon(QWidget):

    def __init__(self):
        super().__init__()

        self.initializezUI()

    def initializezUI(self):
        self.setGeometry(100, 100, 200, 200)
        self.setWindowTitle('Set Icons Example')
        self.setWindowIcon(QIcon('images/pyqt_logo.png'))

        self.createWidgets()

        self.show()

    def createWidgets(self):
        """
        Set up widgets.
        """
        info_label = QLabel("Click on the button and select a fruit.")

        self.images = [
            "images/1_apple.png",
            "images/2_pineapple.png",
            "images/3_watermelon.png",
            "images/4_banana.png"
        ]
```

```
        self.icon_button = QPushButton(self)
        self.icon_button.setIcon(QIcon(random.choice(self.images)))
        self.icon_button.setIconSize(QSize(60, 60))
        self.icon_button.clicked.connect(self.changeButtonIcon)

        # Create vertical layout and add widgets
        v_box = QVBoxLayout()
        v_box.addWidget(info_label)
        v_box.addWidget(self.icon_button)

        # Set main layout of window
        self.setLayout(v_box)

    def changeButtonIcon(self):
        """
        When the button is clicked, change the icon to one of the images in
        the list.
        """
        self.icon_button.setIcon(QIcon(random.choice(self.images)))
        self.icon_button.setIconSize(QSize(60, 60))

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = ChangeIcon()
    sys.exit(app.exec_())
```

You can see what the application should look like on Windows in Figure 5-3. The application icon normally displayed in the top-left corner is changed to the PyQt logo in the right image. Notice how the application icon is missing in the MacOS version in Figure 5-4.
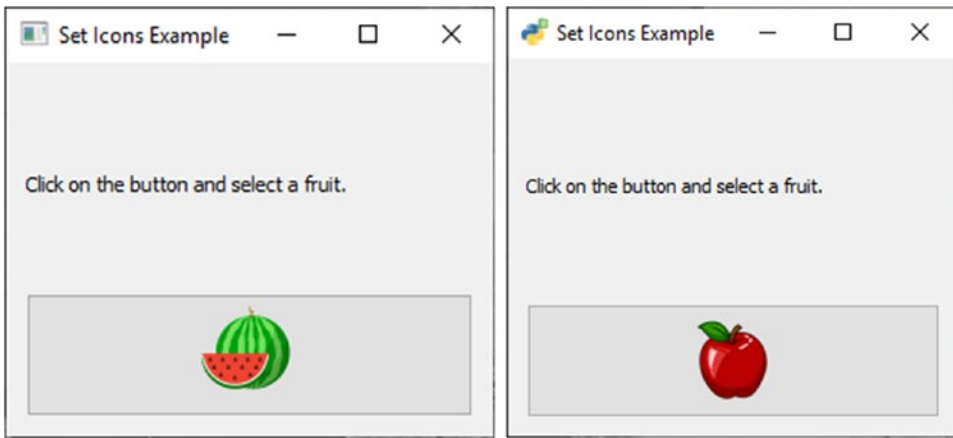
Figure 5-3. *The original application icon in the top-left corner of the window (left) can be set to new a new icon (right) using the setWindowIcon() method. On Windows and Linux systems, changing the icon isn't an issue*
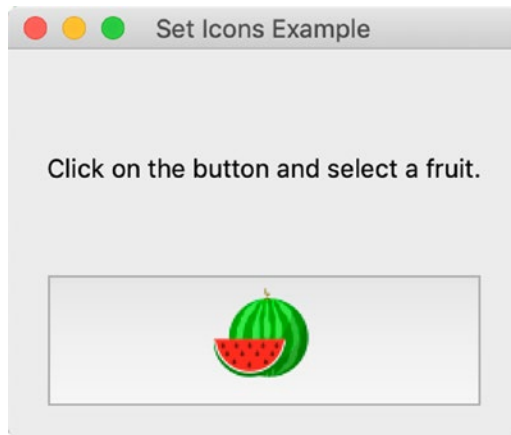
Figure 5-4. *The application icon is not displayed in the title area on MacOS systems*

Explanation

The preceding example contains a simple button that, when clicked, will select an image randomly from the images list.

Setting the main window's application icon can be done by calling the setWindowIcon() method and setting as the argument for QIcon the location of the image file. This can be seen in the following line:

```
self.setWindowIcon(QIcon('images/pyqt_logo.png'))
```

If a widget is created that can display an icon, then calling the `setIcon()` method on that widget will allow you to display an image on it.

```
icon_button.setIcon(QIcon(random.choice(self.images)))
icon_button.setIconSize(QSize(60, 60))
```

Here, the icon for `icon_button` is chosen randomly and passed as an argument to be handled by `QIcon`. Calling the `setIconSize()` method on a widget can be used to change the size of the icon. PyQt will handle the sizing and style of the widget based on your parameters in the main window. The button is then connected to a slot that is used to change the icon.

Finally, the label and button widgets are arranged using `QVBoxLayout` and set as the main window's layout.

Project 5.1 – Rich Text Notepad GUI

For the first project, let's take a look at how to improve the notepad GUI we saw back in Chapter 4. It's important to actually build a complete program to help you to learn how to make your own GUIs from start to finish.

This time we will add a proper menubar with menus and actions. The user will also have the ability to open and save their text, either as HTML or plain text, and edit the text's font, color, or size to give more functionality and creativity to their notes.

Figure 5-5 shows an example of the completed application with text of different sizes, colors, fonts, and highlights.
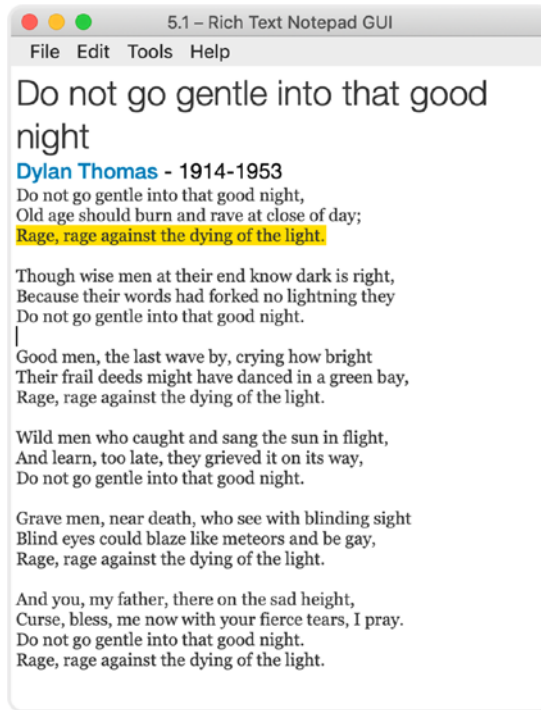
Figure 5-5. *Notepad GUI with menubar and QTextEdit widget*

Design the Rich Text Notepad GUI

Usually before creating interfaces, you should think about and map out what kind of functionality you want your application to have and what kind of widgets you might need in order to achieve those tasks.

For a text editing application, the layout is generally very basic – a menubar at the top of the window with different menus for the various functions and tools, and an area for displaying and editing text. For the text field, we will be using a QTextEdit widget which will also serve as the central widget for the QMainWindow object.

This application will consist of four menus in the menubar – File, Edit, Tools, and Help. Having different menus in the menubar can help to organize actions under different categories as well as help the user to more easily locate actions they want to use. Take a look at Figure 5-6 to see the various menu items that will be included in this project.
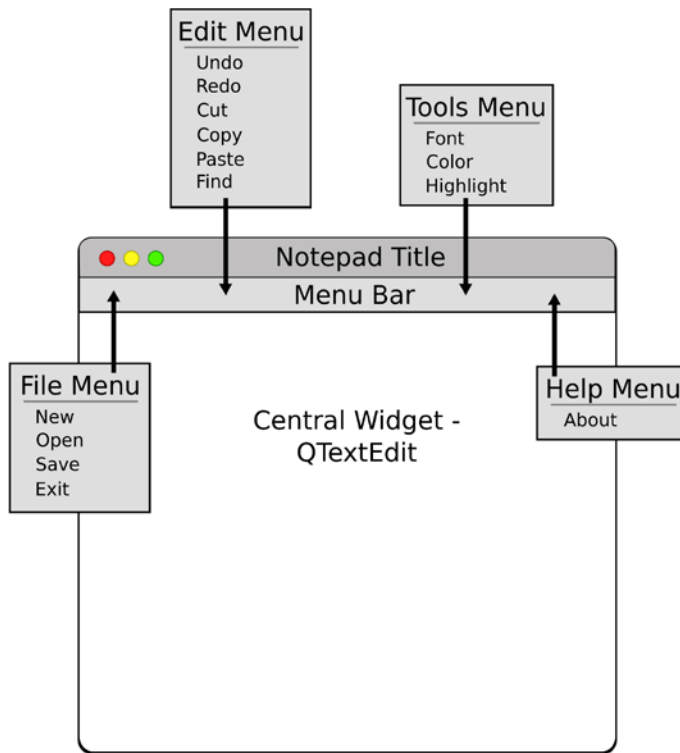
Figure 5-6. *Design showing the layout for the notepad GUI and the different menus and actions*

More Types of Dialog Boxes in PyQt

In this project there are a number of different dialog boxes native to PyQt that are used including QInputDialog, QFileDialog, QFontDialog, QColorDialog, and QMessageBox. Let's take a moment to get familiar with some new types of dialog boxes and find out how to include them in our code.

The QInputDialog Class

QInputDialog is a native dialog box in PyQt that can be used to receive input from the user. The input is a single value that can be a string, a number, or an item from a list.

To create an input dialog and get text from the user:

```
find_text, ok = QInputDialog.getText(self, "Search Text", "Find:")
```

In this example, shown in Figure 5-7, an input dialog object is created by calling `QInputDialog`. The `getText()` method takes a single string input from the user. The second argument, `"Search Text"`, is the title for the dialog and `Find:` is the message displayed in the dialog box. An input dialog returns two values – the input from the user and a Boolean value. If the OK button is clicked, then the `ok` variable is set to `True`.
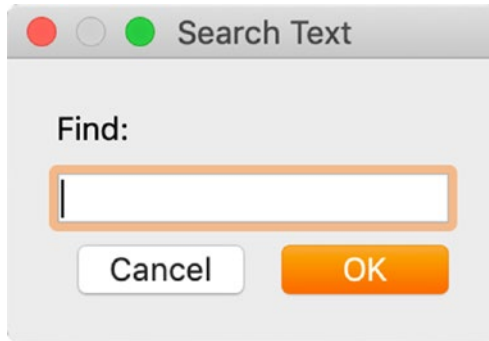
Figure 5-7. *Example of QInputDialog dialog box*

For other types of input, you can use one of the following methods:

- `getMultiLineText()` – Method to get a multiline string from the user
- `getInt()` – Method to get an integer from the user
- `getDouble()` – Method to get a floating-point number from the user
- `getItem()` – Method to let the user select an item from a list of strings

The QFontDialog Class

QFontDialog provides a dialog box that allows the user to select and manipulate different types of fonts. To create a font dialog box and choose a font, use the `getFont()` method. The font dialog that is native to PyQt is shown in Figure 5-8.

```
font, ok = QFontDialog.getFont()
```

The `font` keyword is the particular font returned from `getFont()` and `ok` is a Boolean variable to check whether the user selected a font and clicked the OK button.
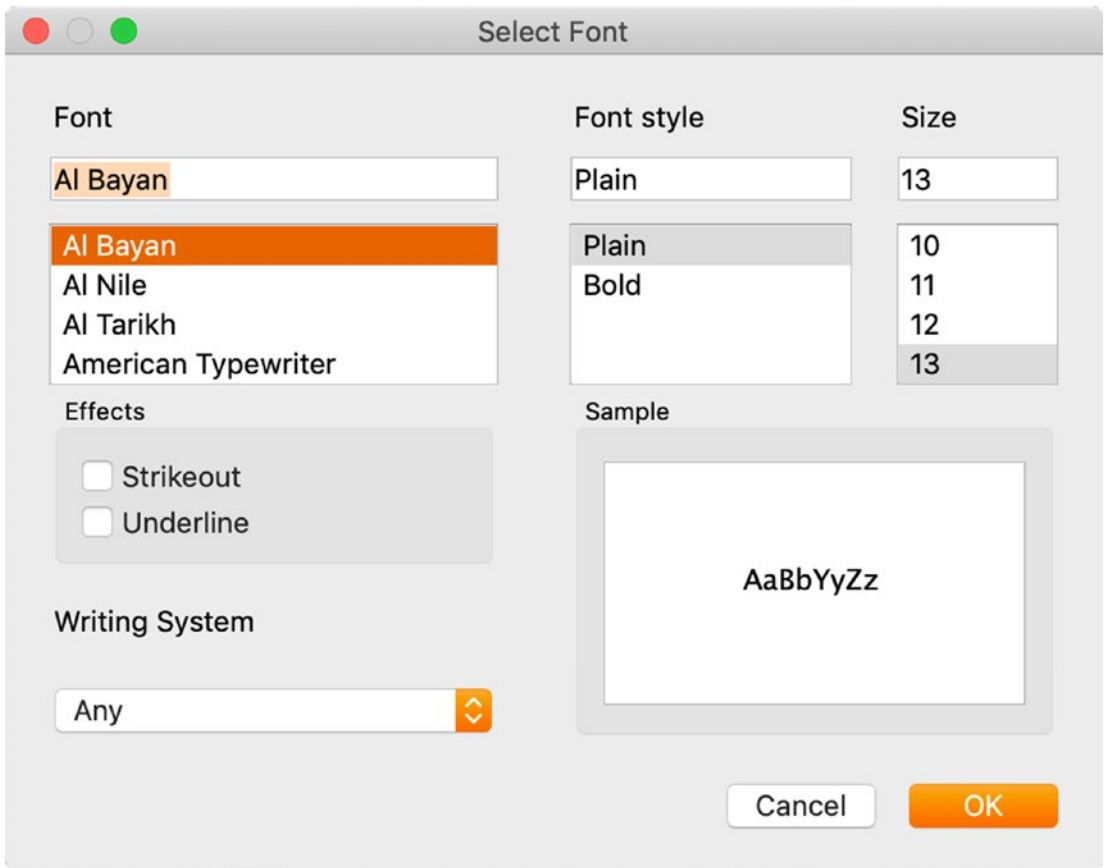
Figure 5-8. *QFontDialog dialog box*

When the user clicks OK, a font is selected. However, if Cancel is clicked, then the initial font is returned. If you have a default font that you would like to use in case the user does not select OK, you could do the following:

```
font, ok = QFontDialog.getFont(QFont("Helvetica", 10), self)
self.text_edit_widget.setCurrentFont(font)
```

In order to change the font if a new one has been chosen, use the setCurrentFont() method and change it to the new font.

The QColorDialog Class

The **QColorDialog** class creates a dialog box for selecting colors like the one in Figure 5-9. Selecting colors can be useful for changing the color of the text, a window's background color, and many other actions.

To create a color dialog box and select a color, use the following line of code:

```
color = QColorDialog.getColor()
```

Then check if the user selected a color and clicked the OK button by using the isValid() method. If so, you could use setTextColor() to change the color of the text or setBackgroundColor() to change the color of the background.

```
if color.isValid():
            self.text_field.setTextColor(color)
```


Figure 5-9. *QColorDialog dialog box*

The About Dialog Box

In many applications you can often find an About item in the menu. Clicking this item will open a dialog box that displays information about the application such as the software's logo, title, latest version number, and other legal information.

The QMessageBox class that we looked at in Chapter 3 also provides an about() method for creating a dialog for displaying a title and text. To create an about dialog box like the one in Figure 5-10, try

```
QMessageBox.about(self, "About Notepad", "Beginner's Practical Guide to
PyQt\n\nProject 5.1 - Notepad GUI")
```

You can also display an application icon in the window. If an icon is not provided, the about() method will try and find one from parent widget. To provide an icon, call the setWindowIcon() method on the QApplication object in the program's main() method.

```
app.setWindowIcon(QIcon("images/app_logo.png"))
```
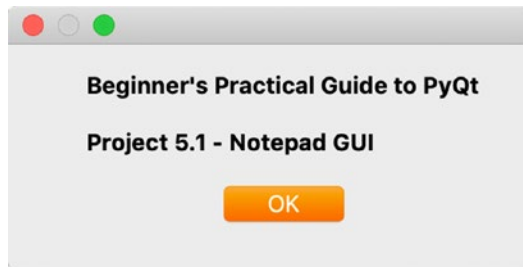

Figure 5-10. *Example About dialog box from the notepad GUI*

Rich Text Notepad GUI Solution

The QTextEdit widget already provides functionality for writing in either plain text or rich text formats. In this program, you will explore how to use the different methods of QTextEdit, such undo() and redo(), as well as the different dialog classes to create a notepad application. This program also allows you to save your text in either plain text format or HTML format if you want to preserve the rich text (Listing 5-3).

Listing 5-3. Rich text notepad GUI code

```python
# richtext_notepad.py
#Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QAction,
QMessageBox, QTextEdit, QFileDialog, QInputDialog, QFontDialog,
QColorDialog)
from PyQt5.QtGui import QIcon, QTextCursor, QColor
from PyQt5.QtCore import Qt

class Notepad(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 400, 500)
        self.setWindowTitle('5.1 - Rich Text Notepad GUI')
        self.createNotepadWidget()
        self.notepadMenu()

        self.show()

    def createNotepadWidget(self):
        """
        Set the central widget for QMainWindow, which is the QTextEdit
        widget for the notepad.
        """
        self.text_field = QTextEdit()
        self.setCentralWidget(self.text_field)
```

```python
def notepadMenu(self):
    """
    Create menu for notepad GUI
    """
    # Create actions for file menu
    new_act = QAction(QIcon('images/new_file.png'), 'New', self)
    new_act.setShortcut('Ctrl+N')
    new_act.triggered.connect(self.clearText)

    open_act = QAction(QIcon('images/open_file.png'), 'Open', self)
    open_act.setShortcut('Ctrl+O')
    open_act.triggered.connect(self.openFile)

    save_act = QAction(QIcon('images/save_file.png'), 'Save', self)
    save_act.setShortcut('Ctrl+S')
    save_act.triggered.connect(self.saveToFile)

    exit_act = QAction(QIcon('images/exit.png'), 'Exit', self)
    exit_act.setShortcut('Ctrl+Q')
    exit_act.triggered.connect(self.close)

    # Create actions for edit menu
    undo_act = QAction(QIcon('images/undo.png'),'Undo', self)
    undo_act.setShortcut('Ctrl+Z')
    undo_act.triggered.connect(self.text_field.undo)

    redo_act = QAction(QIcon('images/redo.png'),'Redo', self)
    redo_act.setShortcut('Ctrl+Shift+Z')
    redo_act.triggered.connect(self.text_field.redo)

    cut_act = QAction(QIcon('images/cut.png'),'Cut', self)
    cut_act.setShortcut('Ctrl+X')
    cut_act.triggered.connect(self.text_field.cut)

    copy_act = QAction(QIcon('images/copy.png'),'Copy', self)
    copy_act.setShortcut('Ctrl+C')
    copy_act.triggered.connect(self.text_field.copy)
```

```python
        paste_act = QAction(QIcon('images/paste.png'),'Paste', self)
        paste_act.setShortcut('Ctrl+V')
        paste_act.triggered.connect(self.text_field.paste)

        find_act = QAction(QIcon('images/find.png'), 'Find', self)
        find_act.setShortcut('Ctrl+F')
        find_act.triggered.connect(self.findTextDialog)

        # Create actions for tools menu
        font_act = QAction(QIcon('images/font.png'), 'Font', self)
        font_act.setShortcut('Ctrl+T')
        font_act.triggered.connect(self.chooseFont)

        color_act = QAction(QIcon('images/color.png'), 'Color', self)
        color_act.setShortcut('Ctrl+Shift+C')
        color_act.triggered.connect(self.chooseFontColor)

       highlight_act = QAction(QIcon('images/highlight.png'), 'Highlight', self)
       highlight_act.setShortcut('Ctrl+Shift+H')
       highlight_act.triggered.connect(self.chooseFontBackgroundColor)

        about_act = QAction('About', self)
        about_act.triggered.connect(self.aboutDialog)

        # Create menubar
        menu_bar = self.menuBar()
        menu_bar.setNativeMenuBar(False)

        # Create file menu and add actions
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(new_act)
        file_menu.addSeparator()
        file_menu.addAction(open_act)
        file_menu.addAction(save_act)
        file_menu.addSeparator()
        file_menu.addAction(exit_act)

        # Create edit menu and add actions
        edit_menu = menu_bar.addMenu('Edit')
        edit_menu.addAction(undo_act)
```

```python
        edit_menu.addAction(redo_act)
        edit_menu.addSeparator()
        edit_menu.addAction(cut_act)
        edit_menu.addAction(copy_act)
        edit_menu.addAction(paste_act)
        edit_menu.addSeparator()
        edit_menu.addAction(find_act)

        # Create tools menu and add actions
        tool_menu = menu_bar.addMenu('Tools')
        tool_menu.addAction(font_act)
        tool_menu.addAction(color_act)
        tool_menu.addAction(highlight_act)

        # Create help menu and add actions
        help_menu = menu_bar.addMenu('Help')
        help_menu.addAction(about_act)

    def openFile(self):
        """
        Open a text or html file and display its contents in
        the text edit field.
        """
        file_name, _ = QFileDialog.getOpenFileName(self, "Open File",
            "", "HTML Files (*.html);;Text Files (*.txt)")

        if file_name:
            with open(file_name, 'r') as f:
                notepad_text = f.read()
            self.text_field.setText(notepad_text)
        else:
            QMessageBox.information(self, "Error",
                "Unable to open file.", QMessageBox.Ok)

    def saveToFile(self):
        """
        If the save button is clicked, display dialog asking user if
        they want to save the text in the text edit field to a text file.
```

```python
        """
        file_name, _ = QFileDialog.getSaveFileName(self, 'Save File',
            "","HTML Files (*.html);;Text Files (*.txt)")

        if file_name.endswith('.txt'):
            notepad_text = self.text_field.toPlainText()
            with open(file_name, 'w') as f:
                f.write(notepad_text)
        elif file_name.endswith('.html'):
            notepad_richtext = self.text_field.toHtml()
            with open(file_name, 'w') as f:
                f.write(notepad_richtext)
        else:
            QMessageBox.information(self, "Error",
                "Unable to save file.", QMessageBox.Ok)

    def clearText(self):
        """
        If the new button is clicked, display dialog asking user if
        they want to clear the text edit field or not.
        """
        answer = QMessageBox.question(self, "Clear Text",
            "Do you want to clear the text?", QMessageBox.No | QMessageBox.Yes,
            QMessageBox.Yes)
        if answer == QMessageBox.Yes:
            self.text_field.clear()
        else:
            pass

    def findTextDialog(self):
        """
        Search for text in QTextEdit widget
        """
        # Display input dialog to ask user for text to search for
        find_text, ok = QInputDialog.getText(self, "Search Text", "Find:")

      extra_selections = []
```

```
    # Check to make sure the text can be modified
    if ok and not self.text_field.isReadOnly():
        # set the cursor in the textedit field to the beginning
        self.text_field.moveCursor(QTextCursor.Start)
        color = QColor(Qt.yellow)

        # Look for next occurrence of text
        while(self.text_field.find(find_text)):
            # Use ExtraSelections to mark the text you are
            # searching for as yellow
            selection = QTextEdit.ExtraSelection()
            selection.format.setBackground(color)

            # Set the cursor of the selection
            selection.cursor = self.text_field.textCursor()

            # Add selection to list
            extra_selections.append(selection)

        # Highlight selections in text edit widget
        for i in extra_selections:
            self.text_field.setExtraSelections(extra_selections)

def chooseFont(self):
    """
    Select font for text
    """
    current = self.text_field.currentFont()
    font, ok = QFontDialog.getFont(current, self, options=QFontDialog.
    DontUseNativeDialog)
    if ok:
        self.text_field.setCurrentFont(font) # Use setFont() to set all
        text to one type of font

def chooseFontColor(self):
    """
    Select color for text
    """
```

```
        color = QColorDialog.getColor()
        if color.isValid():
            self.text_field.setTextColor(color)

    def chooseFontBackgroundColor(self):
        """
        Select color for text's background
        """
        color = QColorDialog.getColor()
        if color.isValid():
            self.text_field.setTextBackgroundColor(color)

    def aboutDialog(self):
        """
        Display information about program dialog box
        """
        QMessageBox.about(self, "About Notepad", "Beginner's Practical
        Guide to PyQt\n\nProject 5.1 - Notepad GUI")

# Run program
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = Notepad()
    sys.exit(app.exec_())
```

Your program with its menubar and different menus should look similar to the images in Figure 5-11.
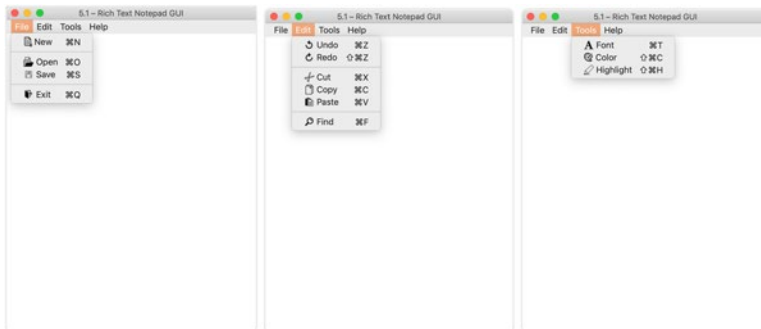
Figure 5-11. *The notepad GUI with its different menus displayed, File menu (left), Edit menu (middle), and Tools menu (right)*

Explanation

There are quite a few classes to import for the notepad application. From the `QtWidgets` module, we need to import `QMainWindow` and `QAction` for creating the menubar and menu items. We also need to include the different PyQt dialog classes such as `QFileDialog` and `QInputDialog`. From `QtGui`, which provides many of the basic classes for creating GUI applications, `QIcon` is used for handling icons, `QTextCursor` can be used to get information about the cursor in text documents, and `QColor` provides methods to create colors in PyQt.

The main window is initialized in the `Notepad` class which inherits from `QMainWindow`. The `createNotepadWidget()` method creates a `QTextEdit` widget and sets it as the central widget for the `QMainWindow` using

```
self.setCentralWidget(self.text_field)
```

Next, the `notepadMenu()` method sets up the menubar object along with the different menu items. The menu for the notepad application, which can be seen in Figure 5-10, contains four menus, `File`, `Edit`, `Tools`, and `Help`. Each menu is given its own menu items for the most part based on guidelines that we have come to expect from applications. For example, a general `File` menu creates actions that allow the user to open, save, import, export, or print files.

The following bit of code shows how to create the action to open a file:

```
open_act = QAction(QIcon('images/open_file.png'), 'Open', self)
open_act.setShortcut('Ctrl+O')
open_act.triggered.connect(self.openFile)
```

The `open_act` object is generated by the `QAction` class. `QIcon` is used to set an icon next to the action's text in the menu. Then the action is given text to display, `Open`. Many of the actions in the notepad program are given a textual shortcut using `setShortcut()`. Finally, we connect the `open_act` signal that is produced when it is triggered to a slot, in this case the `openFile()` method. Other actions are created in a similar manner.

QTextEdit already has predefined slots, such as `cut()`, `copy()`, and `paste()`, to interact with text. For most of the actions in the `Edit` menu, their signals are connected to these special slots rather than creating new ones.

Once all the actions are defined, the menu_bar is created and the different menus are created by using the addMenu() method.

```
file_menu = menu_bar.addMenu('File')
```

Each of the actions is added to a menu by calling the addAction() method on the appropriate menu. To add a divider between categories in a menu, use addSeparator().

```
file_menu.addAction(new_act)
file_menu.addSeparator()
```

There are a number of functions that are called on when a menu item is clicked. Each one of them opens a dialog box and returns some kind of input from the user, such as a new file, text or background color, or a keyword from a text search using the QInputDialog class.

Project 5.2 – Simple Photo Editor GUI

With the introduction of smartphones that have the latest technology to take amazing photos, more pictures are taken and modified every day. However, not every picture is perfect as soon as it is taken and technology also gives us tools to edit those images to our liking. Some photo editors are very simple, allowing the user to rotate, crop, or add text to images. Others let the user change the contrast and exposure, reduce noise, or even add special effects.

In the following project, we will be taking a look at how to create a basic image editor, Figure 5-12, that can give you a foundation to build your own application.
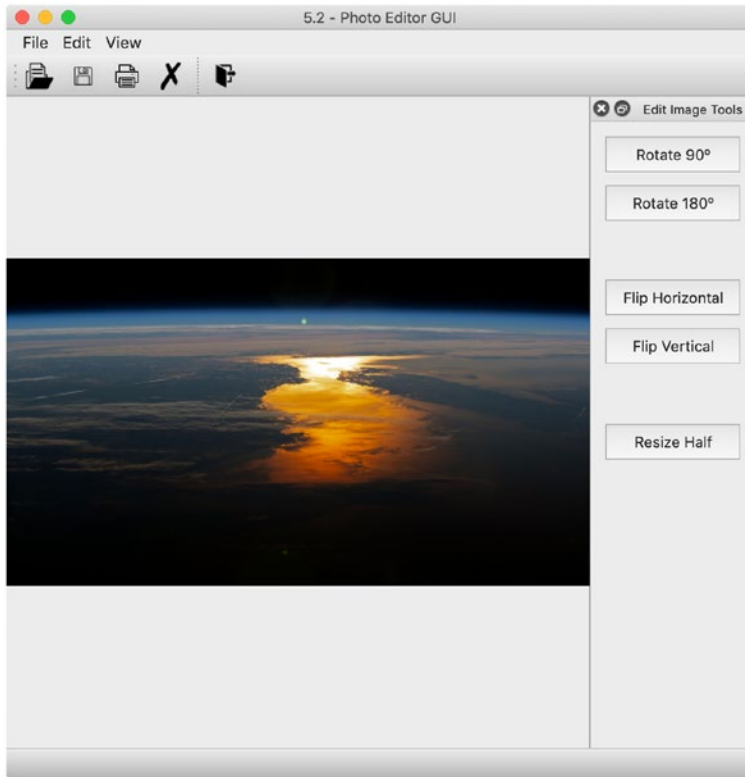
Figure 5-12. *Photo editor GUI displaying the menubar at the top, the toolbar with icons underneath the menubar, the central widget which displays the image, the status bar at the bottom, and the dock widget on the right containing simple tools for editing the photo. Earth photo from* `nasa.org`

Design the Photo Editor GUI

Similar to Project 5.1, this GUI will also have a menubar that will contain various menus – File, Edit, and View. The layout for this project can be seen in Figure 5-13. Under the menubar is the toolbar created using the QToolBar class which contains icons that represent actions the user can take such as open a file, save a file, and print. This project will also introduce the QDockWidget class for creating widgets that can be docked inside the main window or left floating, the QStatusBar class for displaying information to the user, and checkable menu items which we will use to hide or show the dock widget.
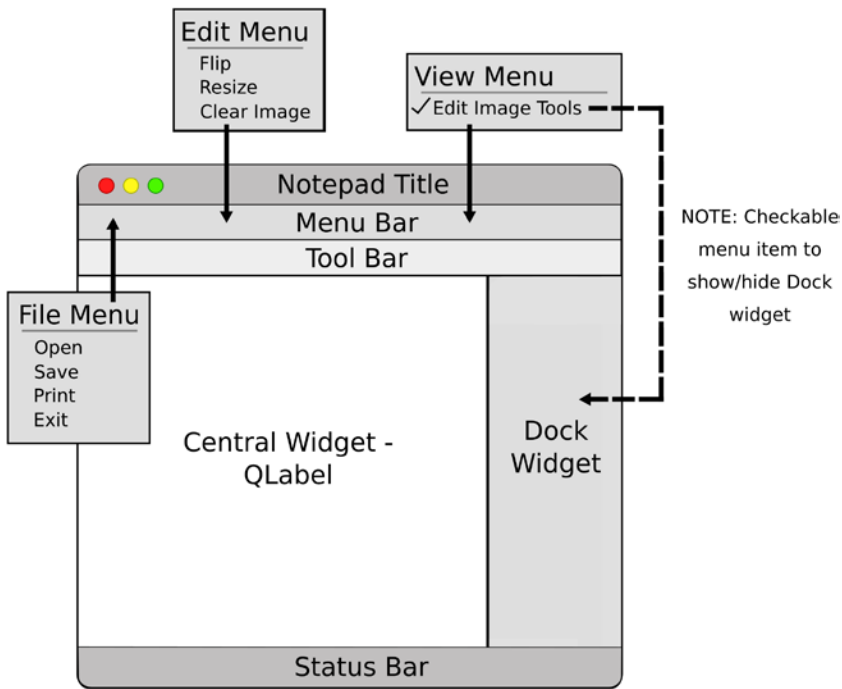
Figure 5-13. *Layout for the photo editor GUI. The main window is much busier than before containing a toolbar, a dock widget, and a status bar*

QDockWidget, QStatusBar, and More

Let's take a look at some of the important features that will be introduced in the photo editor program:

- The QDockWidget class

- The QStatusBar class

- Creating submenus

- Creating checkable menu items

Listing 5-4 creates a more detailed GUI framework that demonstrates these concepts.

Listing 5-4. Code to demonstrate how to create dock widgets, status bars, and toolbars

```
# menu_framework2.py
# Import necessary modules
import sys
```

```python
from PyQt5.QtWidgets import (QApplication, QMainWindow, QStatusBar,
QAction, QTextEdit, QToolBar, QDockWidget)
from PyQt5.QtCore import Qt, QSize
from PyQt5.QtGui import QIcon

class BasicMenu(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 350, 350) # x, y, width, height
        self.setWindowTitle('Basic Menu Example 2')

        # Set central widget for main window
        self.setCentralWidget(QTextEdit())

        self.createMenu()
        self.createToolBar()
        self.createDockWidget()

        self.show()

    def createMenu(self):
        """
        Create menubar and menu actions
        """
        # Create actions for file menu
        self.exit_act = QAction(QIcon('images/exit.png'), 'Exit', self)
        self.exit_act.setShortcut('Ctrl+Q')
        self.exit_act.setStatusTip('Quit program')
        self.exit_act.triggered.connect(self.close)
```

```python
        # Create actions for view menu
        full_screen_act = QAction('Full Screen', self, checkable=True)
        full_screen_act.setStatusTip('Switch to full screen mode')
        full_screen_act.triggered.connect(self.switchToFullScreen)

        # Create menubar
        menu_bar = self.menuBar()
        menu_bar.setNativeMenuBar(False)

        # Create file menu and add actions
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(self.exit_act)

        # Create view menu, Appearance submenu, and add actions
        view_menu = menu_bar.addMenu('View')
        appearance_submenu = view_menu.addMenu('Appearance')
        appearance_submenu.addAction(full_screen_act)

        # Display info about tools, menu, and view in the status bar
        self.setStatusBar(QStatusBar(self))

    def createToolBar(self):
        """
        Create toolbar for GUI
        """
        # Set up toolbar
        tool_bar = QToolBar("Main Toolbar")
        tool_bar.setIconSize(QSize(16, 16))
        self.addToolBar(tool_bar)

        # Add actions to toolbar
        tool_bar.addAction(self.exit_act)

    def createDockWidget(self):
        """
        Create dock widget
```

```
        """
        # Set up dock widget
        dock_widget = QDockWidget()
        dock_widget.setWindowTitle("Example Dock")
        dock_widget.setAllowedAreas(Qt.AllDockWidgetAreas)

        # Set main widget for the dock widget
        dock_widget.setWidget(QTextEdit())

        # Set initial location of dock widget in main window
        self.addDockWidget(Qt.LeftDockWidgetArea, dock_widget)

    def switchToFullScreen(self, state):
        """
        If state is True, then display the main window in full screen.
        Otherwise, return the window to normal.
        """
        if state:
            self.showFullScreen()
        else:
            self.showNormal()

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = BasicMenu()
    sys.exit(app.exec_())
```

Your GUI created from this program should look similar to the one in Figure 5-14.

115
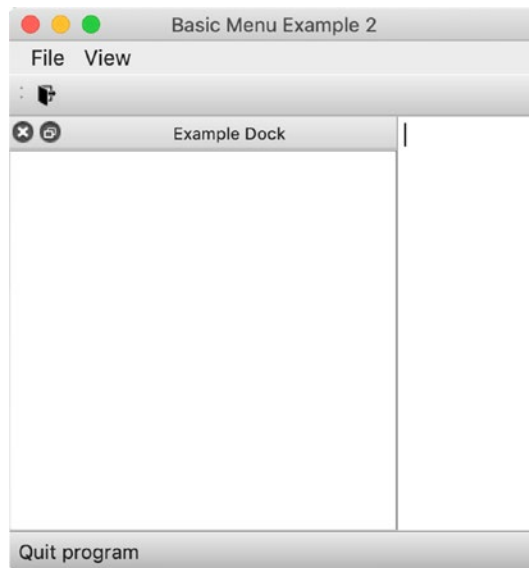
Figure 5-14. *Framework program for creating GUIs with toolbars, status bars, and dock widgets. The status bar on the bottom displays the text "Quit program" when the mouse hovers over the Exit icon in the toolbar*

Explanation

A few new classes from the QtWidgets module are imported including QStatusBar, QToolBar, and QDockWidget. Getting to learn a little bit about these classes will be useful for creating more complex GUIs.

The QStatusBar Class

At the bottom of the GUI in Figure 5-14, there is a horizontal bar with the text "Quit program" displayed inside of it. This bar is known as the status bar as is created from the **QStatusBar** class. Sometimes an icon's or menu item's function is not explicitly understood. This widget is very useful for displaying extra information to the user about the capabilities of an action.

To create a status bar object, you can use the setStatusBar() method which is part of the QMainWindow class. To create an empty status bar, pass QStatusBar as an argument.

```
self.setStatusBar(QStatusBar(self))
```

The first time this method is called, it creates the status bar, and following calls will return the status bar object.

In order to display a message in the status bar when the mouse hovers over an icon, you need to call the setStatusTip() method on an action object. For example:

```
exit_act.setStatusTip('Quit program')
```

will display the text "Quit program" when the mouse is over the exit_act icon or menu command.

To display text in the status bar when the program begins or when a function is called, use the showMessage() method.

```
self.statusBar().showMessage('Welcome back!')
```

The QToolBar Class

When the user is performing a number of routine tasks, having to open up the menu to select an action multiple times can become tedious. Luckily, the **QToolBar** class provides ways to create a toolbar with icons, text, or standard Qt widgets for quick access to frequently used commands.

Toolbars are generally located under the menubar like in Figure 5-14, but can also be placed vertically or at the bottom of the main window above the status bar. Refer to the image in Figure 5-2 for an idea of arranging the different widgets in the main window.

A GUI can only have one menubar but it can have multiple toolbars. To create a toolbar object, create an instance of the QToolBar class and give it a title and then add it to the main window using QMainWindow's addToolBar() method.

```
tool_bar = QToolBar("Main Toolbar")
tool_bar.setIconSize(QSize(16, 16))
self.addToolBar(tool_bar)
```

You should set the size of the icons in the toolbar using the setIconSize() method with QSize() to avoid extra padding when PyQt tries to figure out the arrangement by itself.

To add an action to the toolbar, use addAction():

```
tool_bar.addAction(self.exit_act)
```

If you need to add widgets in your toolbar, you should also use QAction to take advantage of the classes' ability to handle multiple interface elements.

The QDockWidget Class

The **QDockWidget** class is used to create detachable or floating tool palettes or widget panels. Dock widgets are secondary windows that provide additional functionality to GUI windows.

To create the dock widget object, create an instance of `QDockWidget` and set the widget's title using the `setWindowTitle()` method.

```
dock_widget = QDockWidget()
dock_widget.setWindowTitle("Example Dock")
```

When the dock widget is docked inside of the main window, PyQt handles the resizing of the dock window and the central widget. You can also specify the areas you want the dock to be placed in the main window using `setAllowedAreas()`.

```
dock_widget.setAllowedAreas(Qt.AllDockWidgetAreas)
```

In the preceding line of code, the dock widget can be placed on any of the four sides of the window. To limit the allowable dock areas, use the following Qt methods:

- `LeftDockWidgetArea`
- `RightDockWidgetArea`
- `TopDockWidgetArea`
- `BottomDockWidgetArea`

The dock widget can act as a parent for a single widget using `setWidget()`.

```
dock_widget.setWidget(QTextEdit())
```

In order to place multiple widgets inside the dock, you could use a single QWidget as the parent for multiple child widgets and arrange them using one of the layout mangers from Chapter 4. Then, pass that QWidget as the argument to `setWidget()`.

Finally, to set the initial location of the dock widget in the main window, use

```
self.addDockWidget(Qt.LeftDockWidgetArea, dock_widget)
```

In this application if the dock widget is closed, we cannot get it back. In the "Photo Editor GUI Solution," we will take a look at how to use checkable menu items to hide or show the dock widget.

Creating Submenus with Checkable Menu Items

When an application becomes very complex and filled with actions, its menus can also begin to turn into a cluttered mess. Using **submenus**, we can organize similar categories together and simplify the menu system. Figure 5-15 displays an example of a submenu.

Similar to creating a regular menu, use the addMenu() method to create submenus.

```
view_menu = menu_bar.addMenu('View')
appearance_submenu = view_menu.addMenu('Appearance')
appearance_submenu.addAction(full_screen_act)
```

Here we first create the View menu and add it to the menubar. The appearance_ submenu is then created and added to the View menu. Don't forget to also add an action to the submenu using the addAction() method.

The appearance_submenu in the example has a full_screen_act action added to it that allows the user to switch between full screen and normal screen modes. Menu items can also be created so that they act just like switches, being able to be turned on and off. To set an action as checkable, include the option checkable=True in the QAction parameters.

```
full_screen_act = QAction('Full Screen', self, checkable=True)
```

Then, when the action is clicked, it will send a signal and you can use a slot to check the state of the menu item, whether it is on or off. This could be useful for showing or hiding dock widgets or the status bar.

To make the action checked and active from the start, you can call the trigger() method on the action.
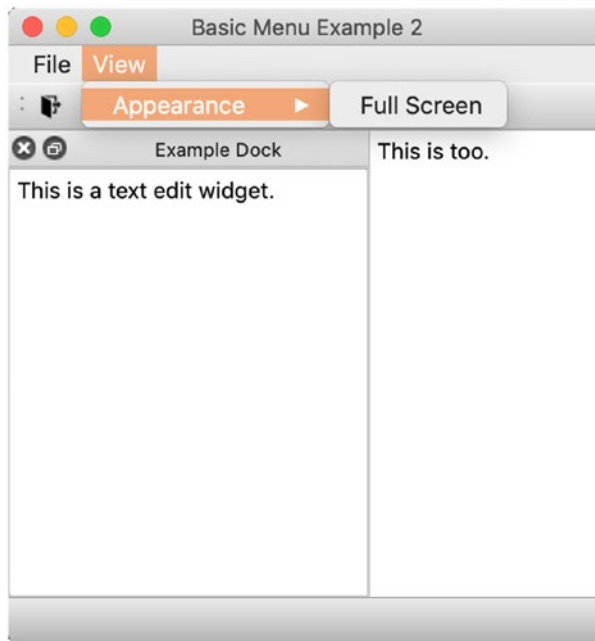
Figure 5-15. *Example submenu that also contains a checkable action to switch between full screen and normal modes*

Photo Editor GUI Solution

Now that we have gone over how to set up the different types of menus, we can finally get started on coding the photo editor application (Listing 5-5).

Listing 5-5. Photo editor code

```
# photo_editor.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QLabel,
QAction, QFileDialog, QDesktopWidget, QMessageBox, QSizePolicy, QToolBar,
QStatusBar, QDockWidget, QVBoxLayout, QPushButton)
from PyQt5.QtGui import QIcon, QPixmap, QTransform, QPainter
from PyQt5.QtCore import Qt, QSize, QRect
from PyQt5.QtPrintSupport import QPrinter, QPrintDialog
```

```python
class PhotoEditor(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setFixedSize(650, 650)
        self.setWindowTitle('5.2 - Photo Editor GUI')
        self.centerMainWindow()
        self.createToolsDockWidget()
        self.createMenu()
        self.createToolBar()
        self.photoEditorWidgets()

        self.show()

    def createMenu(self):
        """
        Create menu for photo editor GUI
        """
        # Create actions for file menu
        self.open_act = QAction(QIcon('images/open_file.png'),"Open", self)
        self.open_act.setShortcut('Ctrl+O')
        self.open_act.setStatusTip('Open a new image')
        self.open_act.triggered.connect(self.openImage)

        self.save_act = QAction(QIcon('images/save_file.png'),"Save", self)
        self.save_act.setShortcut('Ctrl+S')
        self.save_act.setStatusTip('Save image')
        self.save_act.triggered.connect(self.saveImage)

        self.print_act = QAction(QIcon('images/print.png'), "Print", self)
        self.print_act.setShortcut('Ctrl+P')
        self.print_act.setStatusTip('Print image')
```

```python
        self.print_act.triggered.connect(self.printImage)
        self.print_act.setEnabled(False)

        self.exit_act = QAction(QIcon('images/exit.png'), 'Exit', self)
        self.exit_act.setShortcut('Ctrl+Q')
        self.exit_act.setStatusTip('Quit program')
        self.exit_act.triggered.connect(self.close)

        # Create actions for edit menu
        self.rotate90_act = QAction("Rotate 90°", self)
        self.rotate90_act.setStatusTip('Rotate image 90° clockwise')
        self.rotate90_act.triggered.connect(self.rotateImage90)

        self.rotate180_act = QAction("Rotate 180°", self)
        self.rotate180_act.setStatusTip('Rotate image 180° clockwise')
        self.rotate180_act.triggered.connect(self.rotateImage180)

        self.flip_hor_act = QAction("Flip Horizontal", self)
        self.flip_hor_act.setStatusTip('Flip image across horizontal axis')
        self.flip_hor_act.triggered.connect(self.flipImageHorizontal)

        self.flip_ver_act = QAction("Flip Vertical", self)
        self.flip_ver_act.setStatusTip('Flip image across vertical axis')
        self.flip_ver_act.triggered.connect(self.flipImageVertical)

        self.resize_act = QAction("Resize Half", self)
        self.resize_act.setStatusTip('Resize image to half the original size')
        self.resize_act.triggered.connect(self.resizeImageHalf)

        self.clear_act = QAction(QIcon('images/clear.png'), "Clear Image", self)
        self.clear_act.setShortcut("Ctrl+D")
        self.clear_act.setStatusTip('Clear the current image')
        self.clear_act.triggered.connect(self.clearImage)

        # Create menubar
        menu_bar = self.menuBar()
        menu_bar.setNativeMenuBar(False)
```

```python
        # Create file menu and add actions
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(self.open_act)
        file_menu.addAction(self.save_act)
        file_menu.addSeparator()
        file_menu.addAction(self.print_act)
        file_menu.addSeparator()
        file_menu.addAction(self.exit_act)

        # Create edit menu and add actions
        edit_menu = menu_bar.addMenu('Edit')
        edit_menu.addAction(self.rotate90_act)
        edit_menu.addAction(self.rotate180_act)
        edit_menu.addSeparator()
        edit_menu.addAction(self.flip_hor_act)
        edit_menu.addAction(self.flip_ver_act)
        edit_menu.addSeparator()
        edit_menu.addAction(self.resize_act)
        edit_menu.addSeparator()
        edit_menu.addAction(self.clear_act)

        # Create view menu and add actions
        view_menu = menu_bar.addMenu('View')
        view_menu.addAction(self.toggle_dock_tools_act)

        # Display info about tools, menu, and view in the status bar
        self.setStatusBar(QStatusBar(self))

    def createToolBar(self):
        """
        Create toolbar for photo editor GUI
        """
        tool_bar = QToolBar("Photo Editor Toolbar")
        tool_bar.setIconSize(QSize(24,24))
        self.addToolBar(tool_bar)
```

```
        # Add actions to toolbar
        tool_bar.addAction(self.open_act)
        tool_bar.addAction(self.save_act)
        tool_bar.addAction(self.print_act)
        tool_bar.addAction(self.clear_act)
        tool_bar.addSeparator()
        tool_bar.addAction(self.exit_act)

    def createToolsDockWidget(self):
        """
        Use View -> Edit Image Tools menu and click the dock widget on or off.
        Tools dock can be placed on the left or right of the main window.
        """
        # Set up QDockWidget
        self.dock_tools_view = QDockWidget()
        self.dock_tools_view.setWindowTitle("Edit Image Tools")
        self.dock_tools_view.setAllowedAreas(Qt.LeftDockWidgetArea |
            Qt.RightDockWidgetArea)

        # Create container QWidget to hold all widgets inside dock widget
        self.tools_contents = QWidget()

        # Create tool push buttons
        self.rotate90 = QPushButton("Rotate 90°")
        self.rotate90.setMinimumSize(QSize(130, 40))
        self.rotate90.setStatusTip('Rotate image 90° clockwise')
        self.rotate90.clicked.connect(self.rotateImage90)

        self.rotate180 = QPushButton("Rotate 180°")
        self.rotate180.setMinimumSize(QSize(130, 40))
        self.rotate180.setStatusTip('Rotate image 180° clockwise')
        self.rotate180.clicked.connect(self.rotateImage180)

        self.flip_horizontal = QPushButton("Flip Horizontal")
        self.flip_horizontal.setMinimumSize(QSize(130, 40))
        self.flip_horizontal.setStatusTip('Flip image across horizontal axis')
        self.flip_horizontal.clicked.connect(self.flipImageHorizontal)
```

```
        self.flip_vertical = QPushButton("Flip Vertical")
        self.flip_vertical.setMinimumSize(QSize(130, 40))
        self.flip_vertical.setStatusTip('Flip image across vertical axis')
        self.flip_vertical.clicked.connect(self.flipImageVertical)

        self.resize_half = QPushButton("Resize Half")
        self.resize_half.setMinimumSize(QSize(130, 40))
       self.resize_half.setStatusTip('Resize image to half the original size')
        self.resize_half.clicked.connect(self.resizeImageHalf)

        # Set up vertical layout to contain all the push buttons
        dock_v_box = QVBoxLayout()
        dock_v_box.addWidget(self.rotate90)
        dock_v_box.addWidget(self.rotate180)
        dock_v_box.addStretch(1)
        dock_v_box.addWidget(self.flip_horizontal)
        dock_v_box.addWidget(self.flip_vertical)
        dock_v_box.addStretch(1)
        dock_v_box.addWidget(self.resize_half)
        dock_v_box.addStretch(6)

        # Set the main layout for the QWidget, tools_contents,
        # then set the main widget of the dock widget
        self.tools_contents.setLayout(dock_v_box)
        self.dock_tools_view.setWidget(self.tools_contents)

        # Set initial location of dock widget
        self.addDockWidget(Qt.RightDockWidgetArea, self.dock_tools_view)

        # Handles the visibility of the dock widget
        self.toggle_dock_tools_act = self.dock_tools_view.
        toggleViewAction()

    def photoEditorWidgets(self):
        """
        Set up instances of widgets for photo editor GUI
        """
        self.image = QPixmap()
```

```python
        self.image_label = QLabel()
        self.image_label.setAlignment(Qt.AlignCenter)
        # Use setSizePolicy to specify how the widget can be resized,
          horizontally and vertically. Here, the image will stretch
          horizontally, but not vertically.
        self.image_label.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.
        Ignored)

        self.setCentralWidget(self.image_label)

    def openImage(self):
        """
        Open an image file and display its contents in label widget.
        Display error message if image can't be opened.
        """
        image_file, _ = QFileDialog.getOpenFileName(self, "Open Image", "",
            "JPG Files (*.jpeg *.jpg );;PNG Files (*.png);;Bitmap Files
            (*.bmp);;\
                GIF Files (*.gif)")

        if image_file:
            self.image = QPixmap(image_file)

            self.image_label.setPixmap(self.image.scaled(self.image_label.
            size(),
                Qt.KeepAspectRatio, Qt.SmoothTransformation))
        else:
            QMessageBox.information(self, "Error",
                "Unable to open image.", QMessageBox.Ok)

        self.print_act.setEnabled(True)

    def saveImage(self):
        """
        Save the image.
        Display error message if image can't be saved.
        """
```

```python
    image_file, _ = QFileDialog.getSaveFileName(self, "Save Image", "",
        "JPG Files (*.jpeg *.jpg );;PNG Files (*.png);;Bitmap Files
        (*.bmp);;\
            GIF Files (*.gif)")

    if image_file and self.image.isNull() == False:
        self.image.save(image_file)
    else:
        QMessageBox.information(self, "Error",
            "Unable to save image.", QMessageBox.Ok)

def printImage(self):
    """
    Print image.
    """
    # Create printer object and print output defined by the platform
    # the program is being run on.
    # QPrinter.NativeFormat is the default
    printer = QPrinter()
    printer.setOutputFormat(QPrinter.NativeFormat)

    # Create printer dialog to configure printer
    print_dialog = QPrintDialog(printer)

    # If the dialog is accepted by the user, begin printing
    if (print_dialog.exec_() == QPrintDialog.Accepted):
        # Use QPainter to output a PDF file
        painter = QPainter()
        # Begin painting device
        painter.begin(printer)
        # Set QRect to hold painter's current viewport, which
        # is the image_label
        rect = QRect(painter.viewport())
        # Get the size of image_label and use it to set the size
        # of the viewport
        size = QSize(self.image_label.pixmap().size())
        size.scale(rect.size(), Qt.KeepAspectRatio)
```

127

```
        painter.setViewport(rect.x(), rect.y(), size.width(), size.
        height())
        painter.setWindow(self.image_label.pixmap().rect())
        # Scale the image_label to fit the rect source (0, 0)
        painter.drawPixmap(0, 0, self.image_label.pixmap())
        # End painting
        painter.end()

    def clearImage(self):
        """
        Clears current image in QLabel widget
        """
        self.image_label.clear()
        self.image = QPixmap() # reset pixmap so that isNull() = True

    def rotateImage90(self):
        """
        Rotate image 90° clockwise
        """
        if self.image.isNull() == False:
            transform90 = QTransform().rotate(90)
            pixmap = QPixmap(self.image)

            rotated = pixmap.transformed(transform90, mode=Qt.
            SmoothTransformation)

            self.image_label.setPixmap(rotated.scaled(self.image_label.
            size(),
                Qt.KeepAspectRatio, Qt.SmoothTransformation))
            self.image = QPixmap(rotated)
            self.image_label.repaint() # repaint the child widget
        else:
            # No image to rotate
            pass
```

```python
def rotateImage180(self):
    """
    Rotate image 180° clockwise
    """
    if self.image.isNull() == False:
        transform180 = QTransform().rotate(180)
        pixmap = QPixmap(self.image)

        rotated = pixmap.transformed(transform180, mode=Qt.
        SmoothTransformation)

        self.image_label.setPixmap(rotated.scaled(self.image_label.
        size(),
            Qt.KeepAspectRatio, Qt.SmoothTransformation))
        # In order to keep being allowed to rotate the image, set the
          rotated image as self.image
        self.image = QPixmap(rotated)
        self.image_label.repaint() # repaint the child widget
    else:
        # No image to rotate
        pass

def flipImageHorizontal(self):
    """
    Mirror the image across the horizontal axis
    """
    if self.image.isNull() == False:
        flip_h = QTransform().scale(-1, 1)
        pixmap = QPixmap(self.image)

        flipped = pixmap.transformed(flip_h)

        self.image_label.setPixmap(flipped.scaled(self.image_label.
        size(),
            Qt.KeepAspectRatio, Qt.SmoothTransformation))
        self.image = QPixmap(flipped)
        self.image_label.repaint()
```

```python
        else:
            # No image to flip
            pass

    def flipImageVertical(self):
        """
        Mirror the image across the vertical axis
        """
        if self.image.isNull() == False:
            flip_v = QTransform().scale(1, -1)
            pixmap = QPixmap(self.image)

            flipped = pixmap.transformed(flip_v)

            self.image_label.setPixmap(flipped.scaled(self.image_label.
            size(),
                Qt.KeepAspectRatio, Qt.SmoothTransformation))
            self.image = QPixmap(flipped)
            self.image_label.repaint()
        else:
            # No image to flip
            pass

    def resizeImageHalf(self):
        """
        Resize the image to half its current size.
        """
        if self.image.isNull() == False:
            resize = QTransform().scale(0.5, 0.5)
            pixmap = QPixmap(self.image)

            resized = pixmap.transformed(resize)

            self.image_label.setPixmap(resized.scaled(self.image_label.size(),
                Qt.KeepAspectRatio, Qt.SmoothTransformation))
            self.image = QPixmap(resized)
            self.image_label.repaint()
```

```
        else:
            # No image to resize
            pass

    def centerMainWindow(self):
        """
        Use QDesktopWidget class to access information about your screen
        and use it to center the application window.
        """
        desktop = QDesktopWidget().screenGeometry()
        screen_width = desktop.width()
        screen_height = desktop.height()

        self.move((screen_width - self.width()) / 2, (screen_height - self.
        height()) / 2)

# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    app.setAttribute(Qt.AA_DontShowIconsInMenus, True)
    window = PhotoEditor()
    sys.exit(app.exec_())
```

Once complete, your application should look similar to the one in Figure 5-12.

Explanation

The photo editor application imports an assortment of new classes from different modules. From QtWidgets there are two new classes, QDesktopWidget and QSizePolicy. The **QDesktopWidget** class is used to access information about the screen on your computer. We will use it later to learn how to center a GUI on your desktop. The **QSizePolicy** class is used for resizing widgets.

From the QtGui module, we use **QPixmap** for handling images, **QTransform** for performing transformations on images, and **QPainter** which is useful for drawing, painting, and printing.

QRect, from QtCore, is used for creating rectangles. This will be used in the printImage() method.

The **QPrintSupport** module and its classes provide cross-platform support for accessing printers and printing documents.

The window is initialized like before except this time the `setFixedSize()` method is used to set the window's geometry so that it cannot be resized.

All of the menus, actions, icons, and status tips are also created in the `createMenu()` method. One important concept to note is that in this application only the toolbar displays icons, not in the menu. This is set with

```
app.setAttribute(Qt.AA_DontShowIconsInMenus, True)
```

The `File` menu contains the `Open`, `Save`, `Print`, and `Exit` actions. Setting the `setEnabled()` method on the `print_act` to `False` shows a disabled menu item and icon in the toolbar. The `print_act` only becomes enabled after an image is opened in the `openImage()` method.

Handling Images in PyQt

The `Edit` menu contains tools for rotating, flipping, resizing, and clearing images.

```
self.image = QPixmap(image_file)
        self.image_label.setPixmap(self.image.scaled(self.image_label.
size(), Qt.KeepAspectRatio, Qt.SmoothTransformation))
```

When an image file is opened using `QFileDialog`, we create a `QPixmap` object using that image, and then `setPixmap()` is called on the `image_label` to scale and set the image in the `QLabel` widget. Finally, the label is set as the central widget in the main window and resized according to the parameters in the `setSizePolicy()` method. QPixmap and other classes to handle images are covered further in Chapter 9.

The QTransform class provides a number of methods to use transformations on images. The photo editor application provides five actions for manipulating images: Flip 90°, Flip 180°, Flip Horizontal, Flip Vertical, and Resize Half. Figure 5-16 displays an example of an image being rotated 90°.

The tools located in the `Edit` menu could also be located in the toolbar. Instead, they are placed in a dock widget which contains push buttons with the different actions as an example of how to create a dock widget. The dock widget can also be toggled on and off in the `View` menu. To handle when the dock widget is checked or unchecked in the menu or if the user has closed the dock widget with the close button, use the QDockWidget method `toggleViewAction()`.
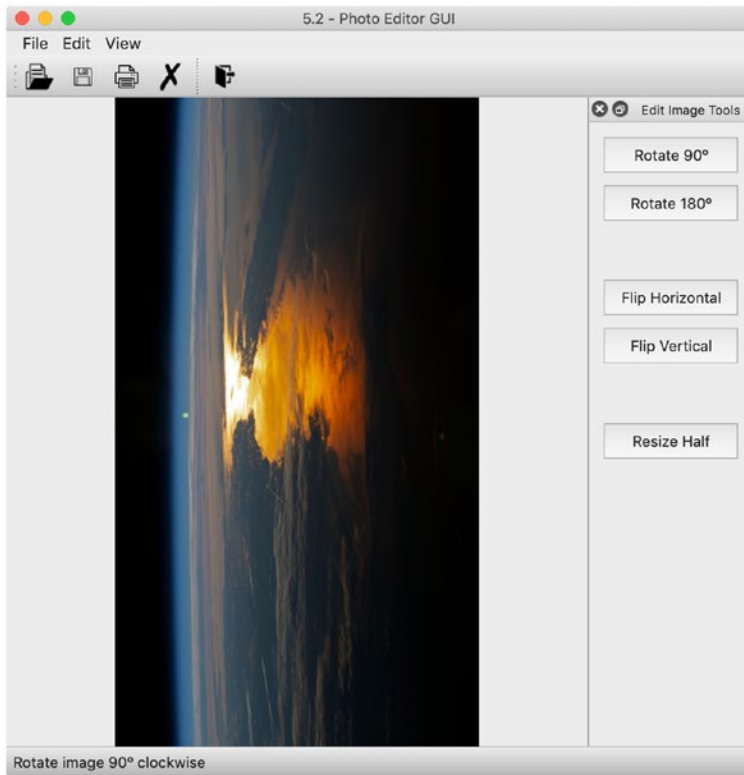
Figure 5-16. *Example of 90° rotation in the photo editor GUI. The image is stretched horizontally to fit in the main window*

The QPrinter Class

If you need to create a printing method for your applications, the photo editor includes a function adopted from the Qt document web site.[1]

Take a look at the code and the comments to see how to use the QPrinter class to set up the printer and QPainter to set up the PDF file to be printed.

Center GUI Application on Your Desktop

The following bit of code shows how to use the **QDesktopWidget** class to find out information about the screen you are using in order to center the widget on the screen when the application starts:

---

[1]https://doc.qt.io/qt-5/qprinter.html#OutputFormat-enum

```
desktop = QDesktopWidget().screenGeometry() # Create QDesktopWidget
screen_width = desktop.width() # Get screen width
screen_height = desktop.height() # Get screen height
# Use absolute positioning to place the GUI in the center of the screen
self.move((screen_width - self.width()) / 2, (screen_height - self.
height()) / 2)
```

Summary

By taking you through some actual examples of programs with working menus, my hope is that you can see how many classes are working together just to make a single application. The examples in this chapter are by no means all that can be done with menus. There are still plenty of other ways to organize the actions and widgets in your own projects including context menus (often referred to as pop-up menus), tabbed widgets with the QToolBox class, stacked widgets using the QStackedWidget class, and more.

Chapter 5 focused on the QMainWindow class for integrating menus easily into GUIs. A menubar consists of several menus, each of which is broken down into several commands. Each of these commands could themselves also be checkable or even submenus. Toolbars are often composed of icons that allow the user to more easily locate commands. The QDockWidget class creates movable and floating widgets that can be used to hold a number of different tools, widgets, or commands. Finally, the status bar created from the QStatusBar class establishes a space to give further textual information about each of the menu items.

The class that acts like the glue to keep track of all the different functions and whether they have been triggered or not is the QAction class. The QAction class manages these actions to ensure that no matter where the action is triggered, whether from a menu, the toolbar, or from shortcut keys, the application can perform the next appropriate action.

In Chapter 6 we will see how to modify the appearance of widgets with style sheets and learn about how to create custom signals in PyQt.

Styling Your GUIs

The GUIs you have created up until now have all focused mainly on function and less on appearance and customization. Creating a layout to organize widgets in a coherent manner is just as important as modifying the look and feel of each and every widget. By choosing the right styles, colors, and fonts, a user can also more easily navigate their way around a user interface.

In this chapter, we will be taking a look at why customizing the look of widgets, windows, and actions is also necessary for designing great GUIs.

In the final section of the chapter, we will take another look at event handling in PyQt and see how we can modify signals and slots to create custom signals to further improve the potential of applications.

Chapter 6 illustrates how to

- Modify the appearance of widgets with `Qt Style Sheets`

- Utilize new Qt widgets and classes, including `QRadioButton`, `QGroupBox`, and `QTabWidget`

- Reimplement event handlers

- Create custom signals using `pyqtSignal` and `QObject`

Changing GUI Appearances with Qt Style Sheets

When you use PyQt, the appearance of your applications is handled by Qt's **QStyle** class. QStyle contains a number of subclasses that imitate the look of the system on which the application is being run. This makes your GUI look like a native MacOS, Linux, or Windows application. Custom styles can be made either by modifying existing QStyle classes, creating new classes, or using Qt Style Sheets.

This chapter will take a look at how to create custom styles by using style sheets. **Qt Style Sheets** provide a technique for customizing the look of widgets. The syntax used in Qt Style Sheets is inspired by HTML Cascading Style Sheets (CSS).

With style sheets, you can customize the look of a number of different widget properties, pseudostates, and subcontrols. Some of the properties that you can modify include background color, font size and font color, border type, width or style, as well as add padding to widgets. **Pseudostates** are used to define special states of a widget, such as when a mouse hovers over a widget or when a widget changes states from active to disabled. **Subcontrols** allow you to access a widget's subelements and change their appearance, location, or other properties. For example, you could change the indicator of a QCheckButton to a different color when it is checked or unchecked.

Customizations can either be applied to individual widgets or to the QApplication object by using `setStyleSheet()`.

Customizing Individual Widget Properties

Let's start by seeing how to apply changes to widgets. The following code changes the background color to blue:

```
line_edit.setStyleSheet("background-color: blue")
```

Colors in a style sheet can be specified using either hexadecimal, RGB, or color keyword formats. To change the foreground color (the text color) of a widget

```
line_edit.setStyleSheet("color: rgb(244, 160, 25") # orange
```

For some widgets as well the main window, you can even set a background image.

```
self.setStyleSheet("background-image: url(images/logo.png)")
```

Now let's take a look at more a detailed example. For the following QLabel widget, we will see how to change the color, the border, some font properties, and the text alignment. The results are shown in Figure 6-1.

```
label = QLabel("Test", self)
label.setStyleSheet("""background-color: skyblue;
        color: white;
        border-style: outset;
        border-width: 3px;
```

```
        border-radius: 5px;
        font: bold 24px 'Times New Roman';
        qproperty-alignment: AlignCenter""")
```

Of course, this is but one example. Each of the different kinds of widgets in Qt has its own parameters that can be customized. For a list of properties that are supported by Qt Style Sheets, refer to Appendix A.
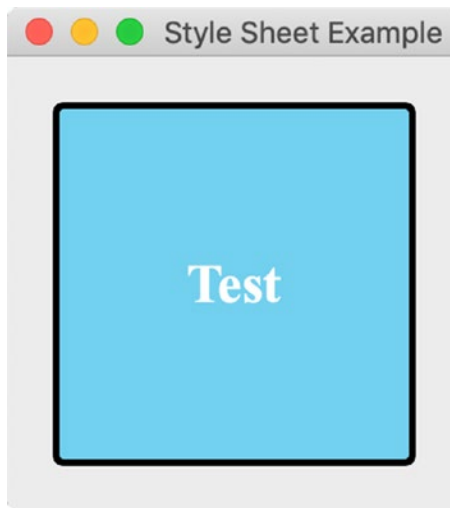
Figure 6-1. *A customized QLabel widget with sky blue background and rounded corners*

Customizing the QApplication Style Sheet

If you have multiple widgets of the same type in an application, you could set each individual widget's style one by one. However, if those widgets all have the same properties, then a much simpler method is to specify all of the modifications at one time.

```
app = QApplication(sys.argv)
app.setStyleSheet("QPushButton{background-color: #C92108}")
```

This will apply a red color to all QPushButton widgets in the GUI. However, if you only want the properties to apply to a specific QPushButton, you can give it an ID selector using setObjectName(). The following excerpt of code shows how to use the ID selector to refer to a particular button. When the button is pressed, a different background color is used.

```
style_sheet = """
    QPushButton#Warning_Button{
        background-color: #C92108;
        border-radius: 10px;
        padding: 6px;
        color: #FFFFFF
    }
    QPushButton#Warning_Button:pressed{
        background-color: #F4B519;
    }
"""
button = QPushButton("Warning!", self)
button.setObjectName("Warning_Button") # Set ID selector

app = QApplication(sys.argv)
app.setStyleSheet(style_sheet) # Set style of QApplication
```

The preceding code also demonstrates how to create a `style_sheet` variable that contains the different properties for each widget. To add a different type of class, simply include the widget type such as QCheckBox followed by the attributes to be changed.

Project 6.1 – Food Ordering GUI

Food delivery service apps are everywhere – on your phone, on the Internet, and even on kiosks when you go to actual restaurants themselves. They simplify the ordering process while also giving the user a feeling of control over their choices, asking us to select our own foods and items as we scroll through a list of organized categories.

These types of GUIs may possibly need to contain hundreds of different items that fit into multiple groups. Rather than just throwing all of the products into the interface and letting the user waste their own time sorting through the items, goods are usually placed into categories often differentiated by tabs. These tabs contain titles for the products that can be found on those corresponding pages, such as Frozen Foods or Fruits/Vegetables.

The GUI in this project allows the user to place an order for a pizza. It lays a foundation for a food ordering application, using tab widgets to organize items onto separate pages. The project also shows how you can use style sheets to give a GUI made using PyQt5 a more visually pleasing appearance. The application can be seen in Figure 6-2.
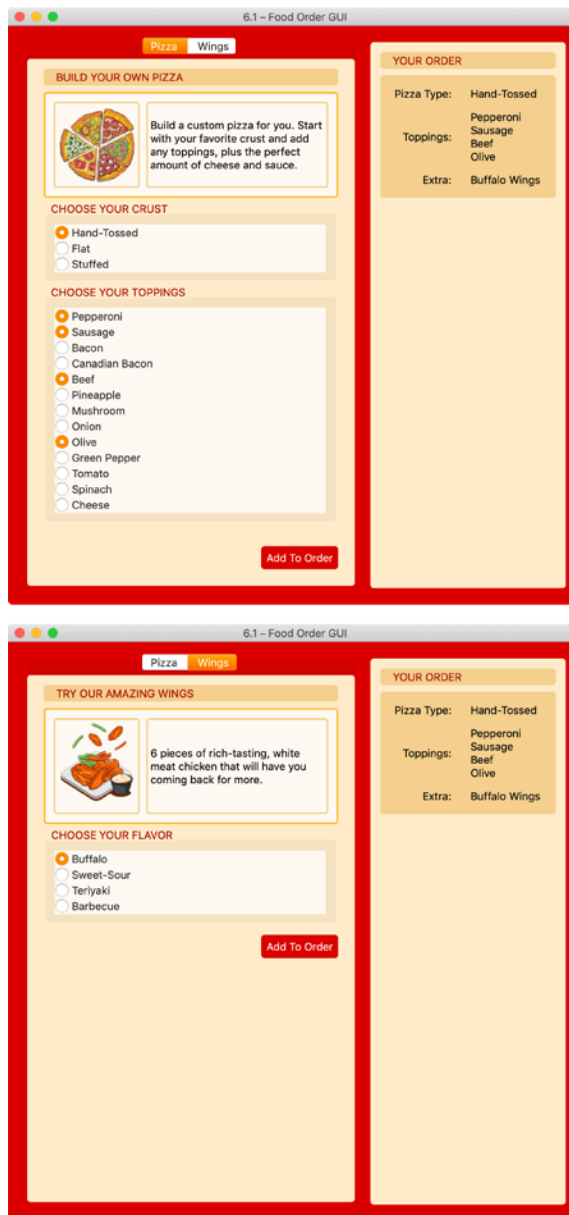
Figure 6-2. *The food ordering GUI. The GUI contains two tabs, Pizza (top) and Wings (bottom), to separate the types of food a customer can see at one time. The choices that can be selected, which are QRadioButton widgets, are separated using QGroupBox widgets. The main window has a red background, and each tab has a tan background. These colors and other styles are created by using a style sheet*

Design the Food Ordering GUI

This application consists of two main tabs (displayed in Figure 6-3), but more could be easily added. Each tab consists of a QWidget that acts as a container for all of the other widgets. The first tab, Pizza, contains an image and text to convey the purpose of the tab to the user. This is followed by two QGroupBox widgets that each consist of a number of QRadioButton widgets. While the radio buttons in the "Crust" group box are mutually exclusive, the ones in the "Toppings" group box are not, so that the user can select multiple options at one time.

The second tab, Wings, is set up in a similar fashion with the "Flavor" radio buttons being mutually exclusive.

At the bottom of each page is an "Add to Order" QPushButton that will update the user's order in the widget on the right-hand side of the window.
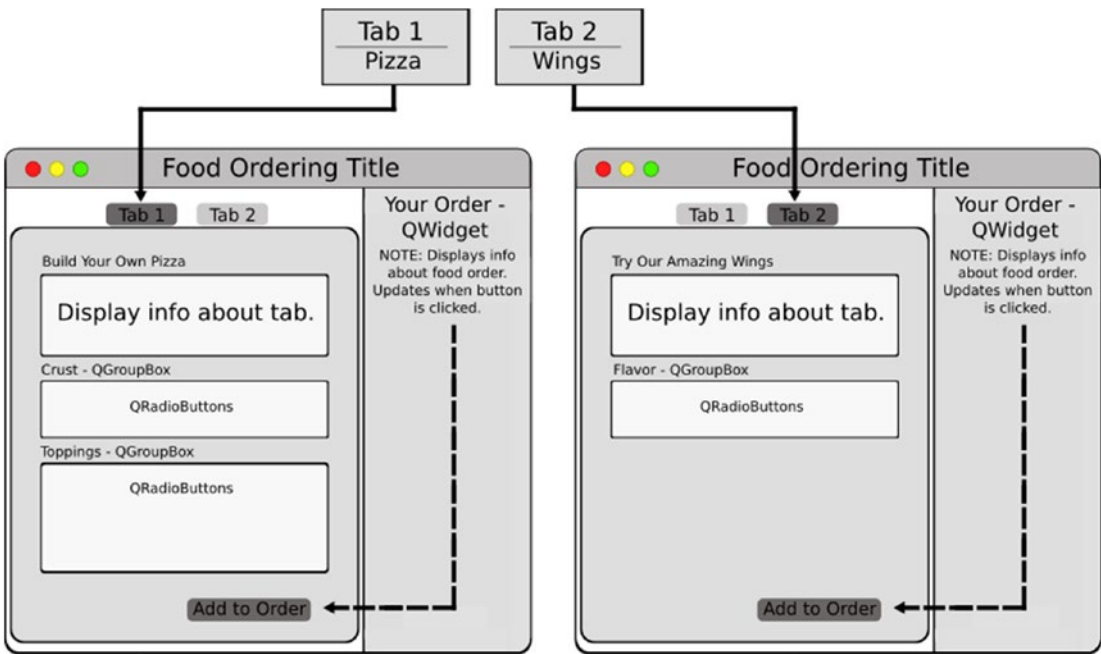
Figure 6-3. *The design for the food ordering GUI*

Before we look at the code for the food ordering GUI, let's take a moment to learn about the new Qt classes in this project – QGroupBox, QRadioButton, and QTabWidget.

The QRadioButton Widget

The **QRadioButton** class allows you to create option buttons that can be switched on when checked or off when unchecked. Each radio button consists of a round button and a corresponding label or icon. Radio buttons are generally used for situations where you need to provide a user with multiple choices, but only one choice can be checked at a time. As the user selects a new radio button, the other radio buttons are unchecked.

When you place multiple radio buttons in a parent widget, those buttons become **auto-exclusive**, meaning they automatically become exclusive members of that group. If one radio button is checked inside of the parent, all of the other buttons will become unchecked. To change this functionality, you can set the setAutoExclusive() attribute to False.

Also, if you want to place multiple exclusive groups of radio buttons into the same parent widget, then use the QButtonGroup class to keep the different groups separate. Refer back to Chapter 4 for information about QButtonGroup.

Radio buttons are similar to the QCheckBox class when emitting signals. A radio button emits the toggled() signal when checked on or off and can be connected to this signal to trigger an action.

An example of creating QRadioButton widgets can be seen in Listing 6-1.

The QGroupBox Class

The **QGroupBox** widget provides a container for grouping other widgets with similar purposes together. A group box has a border with a title on the top. The title can also be checkable so that the child widgets inside the group box can be enabled or disabled when the checkbox is checked or unchecked.

A group box object can contain any kind of widget. Since QGroupBox does not automatically lay out its child widgets, you will need to apply a layout such as QHBoxLayout or QGridLayout. The following snippet of code demonstrates how to create a QGroupBox widget, add two radio buttons, and apply a layout:

```
effects_gb = QGroupBox("Effects") # The title can either be set in the
constructor or with the setTitle() method

# Create instances of radio buttons
effect1_rb = QRadioButton("Strikethrough")
```

```
effect2_rb = QRadioButton("Outline")

# Set up layout and add child widgets to the layout
h_box = QHBoxLayout()
h_box.addWidget(effect1_rb)
h_box.addWidget(effect2_rb)

# Set the layout of the group box
effects_gb.setLayout(h_box)
```

For an example of another type of container in PyQt, check out the QFrame class in Chapter 7.

The QTabWidget Class

Sometimes you may need to organize related information onto separate pages rather than create a cluttered GUI. The **QTabWidget** class provides a tab bar (created from the QTabBar class) with an area under each tab (referred to as a page) to present information and widgets related to each tab. Only one page is displayed at a time, and the user can view a different page by clicking the tab or by using a shortcut (if one is set for the tab).

There are a few different ways to interact with and keep track of the different tabs. For example, if the user switches to a different tab, the currentChanged() signal is emitted. You can also keep track of a current page's index with currentIndex(), or the widget of the current page with currentWidget(). A tab can also be enabled or disabled with the setTabEnabled() method.

Tip   If you want to create an interface with multiple pages, but without the tab bar, then you should consider using a QStackedWidget. However, if you do use QStackedWidget, then you will need to provide some other means to switch between the windows, such as a QComboBox or a QListWidget, since there are no tabs.

The following example creates a simple application that includes QRadioButton, QGroupBox, and QTabWidget and a few other classes. The program shows how to set up a tab widget and organize the other widgets on the different pages.

Listing 6-1. Example that shows how to use QTabWidget, QRadioButton, and QGroupBox classes

```python
# contact_form.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QTabWidget, QLabel,
QRadioButton, QGroupBox, QLineEdit, QHBoxLayout, QVBoxLayout)

class ContactForm(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 400, 300)
        self.setWindowTitle('Contact Form Example')

        self.setupTabs()

        self.show()

    def setupTabs(self):
        """
        Set up tab bar and different tab widgets. Each tab is a QWidget
        that serves as a container for each page.
        """
        # Create tab bar and different tabs
        self.tab_bar = QTabWidget(self)

        self.prof_details_tab = QWidget()
        self.background_tab = QWidget()

        self.tab_bar.addTab(self.prof_details_tab, "Profile Details")
        self.tab_bar.addTab(self.background_tab, "Background")
```

```python
        # Call methods that contain the widgets for each tab
        self.profileDetailsTab()
        self.backgroundTab()

        # Create layout for main window
        main_h_box = QHBoxLayout()
        main_h_box.addWidget(self.tab_bar)

        # Set main window's layout
        self.setLayout(main_h_box)

    def profileDetailsTab(self):
        """
        Create the profile tab. Allows the user enter their name,
        address and select their gender.
        """
        # Set up labels and line edit widgets
        name_label = QLabel("Name")
        name_entry = QLineEdit()

        address_label = QLabel("Address")
        address_entry = QLineEdit()

        # Create group box to contain radio buttons
        sex_gb = QGroupBox("Sex")

        male_rb = QRadioButton("Male")
        female_rb = QRadioButton("Female")

        # Create and set layout for sex_gb widget
        sex_h_box = QHBoxLayout()
        sex_h_box.addWidget(male_rb)
        sex_h_box.addWidget(female_rb)

        sex_gb.setLayout(sex_h_box)

        # Add all widgets to the profile details page layout
        tab_v_box = QVBoxLayout()
        tab_v_box.addWidget(name_label)
        tab_v_box.addWidget(name_entry)
        tab_v_box.addStretch()
```

```
        tab_v_box.addWidget(address_label)
        tab_v_box.addWidget(address_entry)
        tab_v_box.addStretch()
        tab_v_box.addWidget(sex_gb)

        # Set layout for profile details tab
        self.prof_details_tab.setLayout(tab_v_box)

    def backgroundTab(self):
        """
        Create the background tab. The user can select a
        """
        # Set up group box to hold radio buttons
        self.education_gb = QGroupBox("Highest Level of Education")

        # Layout for education_gb
        ed_v_box = QVBoxLayout()

        # Create and add radio buttons to ed_v_box
        education_list = ["High School Diploma", "Associate's Degree",
        "Bachelor's Degree", "Master's Degree", "Doctorate or Higher"]
        for ed in education_list:
            self.education_rb = QRadioButton(ed)
            ed_v_box.addWidget(self.education_rb)
        # Set layout for group box
        self.education_gb.setLayout(ed_v_box)

        # Create and set for background tab
        tab_v_box = QVBoxLayout()
        tab_v_box.addWidget(self.education_gb)

        self.background_tab.setLayout(tab_v_box)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = ContactForm()
    sys.exit(app.exec_())
```

Figure 6-4 shows you how the GUI should look for each tab.
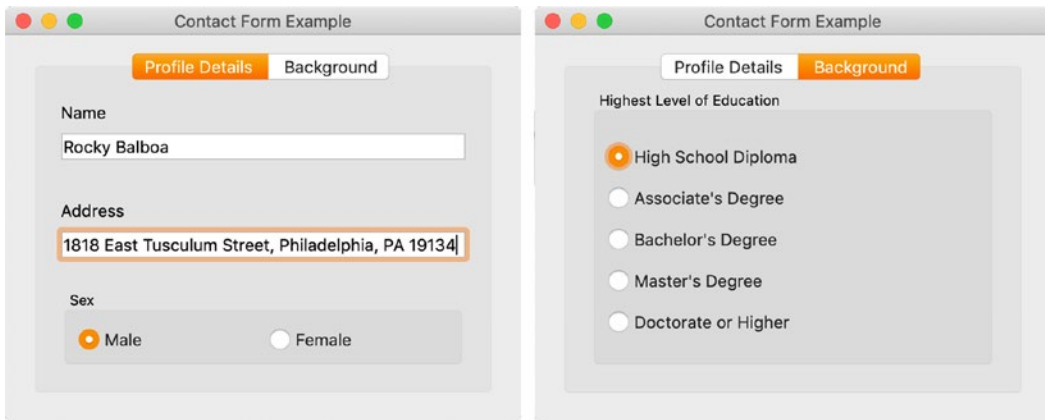
Figure 6-4. *The contact form GUI. The Profile Details tab (left) contains two labels and two line edit widgets as well as a group box with two radio buttons. The Background tab (right) consists of a single group box container with five radio buttons*

Explanation

Let's take a look at how to set up the tab widget and its child widgets in this example.

We begin by importing the necessary classes, including QRadioButton, QTabWidget, and QGroupBox from the QtWidgets module. Next, we set up the ContactForm class and initialize the window's size and title.

The next step is to set up the tab widget and each page in the setupTabs() method. The process to use QTabWidget is to first create an instance of the tab widget. Here we create tab_bar. Then, create a QWidget object for each page in the tab bar. There are two pages for this project, profile_details_tab and background_tab.

Insert the two pages into the tab widget using addTab(). Give each tab an appropriate label.

Finally, create the child widgets for each page and use layouts to arrange them. Two separate methods are created, profileDetailsTab() and backgroundTab(), to organize the two different pages. The labels and line edit widgets are set up like normal. For the QRadioButton objects, they are added to group boxes on their respective pages. In the backgroundTab() method, a for loop is used to instantiate each radio button and add it to the page's layout.

Food Ordering GUI Solution

Now that we have taken a look at the new widgets in this chapter, we can finally move onto the code for the food ordering interface in Listing 6-2.

Listing 6-2. Code for food ordering GUI

```python
# food_order.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QTabWidget, QLabel,
QRadioButton, QButtonGroup, QGroupBox, QPushButton, QVBoxLayout,
QHBoxLayout, QGridLayout)
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import Qt

# Set up style sheet for the entire GUI
style_sheet = """
    QWidget{
        background-color: #C92108;
    }

    QWidget#Tabs{
        background-color: #FCEBCD;
        border-radius: 4px
    }

    QWidget#ImageBorder{
        background-color: #FCF9F3;
        border-width: 2px;
        border-style: solid;
        border-radius: 4px;
        border-color: #FABB4C
    }

    QWidget#Side{
        background-color: #EFD096;
        border-radius: 4px
    }
```

```
QLabel{
    background-color: #EFD096;
    border-width: 2px;
    border-style: solid;
    border-radius: 4px;
    border-color: #EFD096
}

QLabel#Header{
    background-color: #EFD096;
    border-width: 2px;
    border-style: solid;
    border-radius: 4px;
    border-color: #EFD096;
    padding-left: 10px;
    color: #961A07
}

QLabel#ImageInfo{
    background-color: #FCF9F3;
    border-radius: 4px
}

QGroupBox{
    background-color: #FCEBCD;
    color: #961A07
}

QRadioButton{
    background-color: #FCF9F3
}

QPushButton{
    background-color: #C92108;
    border-radius: 4px;
    padding: 6px;
    color: #FFFFFF
}
```

```
    QPushButton:pressed{
        background-color: #C86354;
        border-radius: 4px;
        padding: 6px;
        color: #DFD8D7
    }
"""

class FoodOrderGUI(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(600, 700)
        self.setWindowTitle('6.1 - Food Order GUI')

        self.setupTabsAndLayout()

        self.show()

    def setupTabsAndLayout(self):
        """
        Set up tab bar and different tab widgets.
        Also, create the side widget to display items selected.
        """
        # Create tab bar, different tabs, and set object names
        self.tab_bar = QTabWidget(self)

        self.pizza_tab = QWidget()
        self.pizza_tab.setObjectName("Tabs")
        self.wings_tab = QWidget()
        self.wings_tab.setObjectName("Tabs")
```

```
        self.tab_bar.addTab(self.pizza_tab, "Pizza")
        self.tab_bar.addTab(self.wings_tab, "Wings")

        # Call methods that contain the widgets for each tab
        self.pizzaTab()
        self.wingsTab()

        # Set up side widget which is not part of the tab widget
        self.side_widget = QWidget()
        self.side_widget.setObjectName("Tabs")
        order_label = QLabel("YOUR ORDER")
        order_label.setObjectName("Header")

        items_box = QWidget()
        items_box.setObjectName("Side")
        pizza_label = QLabel("Pizza Type: ")
        self.display_pizza_label = QLabel("")
        toppings_label = QLabel("Toppings: ")
        self.display_toppings_label = QLabel("")
        extra_label = QLabel("Extra: ")
        self.display_wings_label = QLabel("")

        # Set grid layout for objects in side widget
        items_grid = QGridLayout()
        items_grid.addWidget(pizza_label, 0, 0, Qt.AlignRight)
        items_grid.addWidget(self.display_pizza_label, 0, 1)
        items_grid.addWidget(toppings_label, 1, 0, Qt.AlignRight)
        items_grid.addWidget(self.display_toppings_label, 1, 1)
        items_grid.addWidget(extra_label, 2, 0, Qt.AlignRight)
        items_grid.addWidget(self.display_wings_label, 2, 1)
        items_box.setLayout(items_grid)

        # Set main layout for side widget
        side_v_box = QVBoxLayout()
        side_v_box.addWidget(order_label)
        side_v_box.addWidget(items_box)
        side_v_box.addStretch()
        self.side_widget.setLayout(side_v_box)
```

```python
        # Add widgets to main window and set layout
        main_h_box = QHBoxLayout()
        main_h_box.addWidget(self.tab_bar)
        main_h_box.addWidget(self.side_widget)

        self.setLayout(main_h_box)

    def pizzaTab(self):
        """
        Create the pizza tab. Allows the user to select the type of pizza
        and topping using radio buttons.
        """
        # Set up widgets and layouts to display information
        # to the user about the page
        tab_pizza_label = QLabel("BUILD YOUR OWN PIZZA")
        tab_pizza_label.setObjectName("Header")
        description_box = QWidget()
        description_box.setObjectName("ImageBorder")
        pizza_image_path = "images/pizza.png"
        pizza_image = self.loadImage(pizza_image_path)
        pizza_desc = QLabel()
        pizza_desc.setObjectName("ImageInfo")
        pizza_desc.setText("Build a custom pizza for you. Start with your
        favorite crust and add any toppings, plus the perfect amount of
        cheese and sauce.")
        pizza_desc.setWordWrap(True)

        h_box = QHBoxLayout()
        h_box.addWidget(pizza_image)
        h_box.addWidget(pizza_desc)

        description_box.setLayout(h_box)

        # Create group box that will contain crust choices
        crust_gbox = QGroupBox()
        crust_gbox.setTitle("CHOOSE YOUR CRUST")
```

```python
        # The group box is used to group the widgets together,
        # while the button group is used to get information
        # about which radio button is checked
        self.crust_group = QButtonGroup()
        gb_v_box = QVBoxLayout()
        crust_list = ["Hand-Tossed", "Flat", "Stuffed"]
        # Create radio buttons for the different crusts and
        # add to layout
        for cr in crust_list:
            crust_rb = QRadioButton(cr)
            gb_v_box.addWidget(crust_rb)
            self.crust_group.addButton(crust_rb)

        crust_gbox.setLayout(gb_v_box)

        # Create group box that will contain toppings choices
        toppings_gbox = QGroupBox()
        toppings_gbox.setTitle("CHOOSE YOUR TOPPINGS")

        # Set up button group for toppings radio buttons
        self.toppings_group = QButtonGroup()
        gb_v_box = QVBoxLayout()

        toppings_list = ["Pepperoni", "Sausage", "Bacon", "Canadian Bacon",
        "Beef", "Pineapple", "Mushroom", "Onion", "Olive", "Green Pepper",
        "Tomato", "Spinach", "Cheese"]
        # Create radio buttons for the different toppings and
        # add to layout
        for top in toppings_list:
            toppings_rb = QRadioButton(top)
            gb_v_box.addWidget(toppings_rb)
            self.toppings_group.addButton(toppings_rb)
        self.toppings_group.setExclusive(False)

        toppings_gbox.setLayout(gb_v_box)
```

```
    # Create button to add information to side widget
    # when clicked
    add_to_order_button1 = QPushButton("Add To Order")
    add_to_order_button1.clicked.connect(self.displayPizzaInOrder)

    # Create layout for pizza tab (page 1)

    page1_v_box = QVBoxLayout()
    page1_v_box.addWidget(tab_pizza_label)
    page1_v_box.addWidget(description_box)
    page1_v_box.addWidget(crust_gbox)
    page1_v_box.addWidget(toppings_gbox)
    page1_v_box.addStretch()
    page1_v_box.addWidget(add_to_order_button1, alignment=Qt.
    AlignRight)

    self.pizza_tab.setLayout(page1_v_box)

def wingsTab(self):
    # Set up widgets and layouts to display information
    # to the user about the page
    tab_wings_label = QLabel("TRY OUR AMAZING WINGS")
    tab_wings_label.setObjectName("Header")
    description_box = QWidget()
    description_box.setObjectName("ImageBorder")
    wings_image_path = "images/wings.png"
    wings_image = self.loadImage(wings_image_path)
    wings_desc = QLabel()
    wings_desc.setObjectName("ImageInfo")
    wings_desc.setText("6 pieces of rich-tasting, white meat chicken
    that will have you coming back for more.")
    wings_desc.setWordWrap(True)

    h_box = QHBoxLayout()
    h_box.addWidget(wings_image)
    h_box.addWidget(wings_desc)

    description_box.setLayout(h_box)
```

```
        wings_gbox = QGroupBox()
        wings_gbox.setTitle("CHOOSE YOUR FLAVOR")

        self.wings_group = QButtonGroup()
        gb_v_box = QVBoxLayout()
        wings_list = ["Buffalo", "Sweet-Sour", "Teriyaki", "Barbecue"]

        # Create radio buttons for the different flavors and
        # add to layout
        for fl in wings_list:
            flavor_rb = QRadioButton(fl)
            gb_v_box.addWidget(flavor_rb)
            self.wings_group.addButton(flavor_rb)

        wings_gbox.setLayout(gb_v_box)

        # Create button to add information to side widget
        # when clicked
        add_to_order_button2 = QPushButton("Add To Order")
        add_to_order_button2.clicked.connect(self.displayWingsInOrder)

        # Create layout for wings tab (page 2)
        page2_v_box = QVBoxLayout()
        page2_v_box.addWidget(tab_wings_label)
        page2_v_box.addWidget(description_box)
        page2_v_box.addWidget(wings_gbox)
        page2_v_box.addWidget(add_to_order_button2, alignment=Qt.
        AlignRight)
        page2_v_box.addStretch()

        self.wings_tab.setLayout(page2_v_box)

    def loadImage(self, img_path):
        """
        Load and scale images.
        """
```

```
    try:
        with open(img_path):
            image = QLabel(self)
            image.setObjectName("ImageInfo")
            pixmap = QPixmap(img_path)
            image.setPixmap(pixmap.scaled(image.size(),
            Qt.KeepAspectRatioByExpanding, Qt.SmoothTransformation))
            return image
    except FileNotFoundError:
        print("Image not found.")

def collectToppingsInList(self):
    """
    Create list of all checked toppings radio buttons.
    """
    toppings_list = [button.text() for i, button in enumerate(self.
    toppings_group.buttons()) if button.isChecked()]
    return toppings_list

def displayPizzaInOrder(self):
    """
    Collect the text from the radio buttons that are checked on pizza
    page. Display text in side widget.
    checkedButton() returns the buttons that are checked in the
    QButtonGroup.
    """
    try:
        pizza_text = self.crust_group.checkedButton().text()
        self.display_pizza_label.setText(pizza_text)

        toppings = self.collectToppingsInList()
        toppings_str = '\n'.join(toppings)
        self.display_toppings_label.setText(toppings_str)
        self.repaint()
    except AttributeError:
        print("No value selected.")
        pass
```

```python
    def displayWingsInOrder(self):
        """
        Collect the text from the radio buttons that are checked on wings
        page. Display text in side widget.
        """
        try:
            text = self.wings_group.checkedButton().text() + " Wings"
            self.display_wings_label.setText(text)
            self.repaint()
        except AttributeError:
            print("No value selected.")
            pass

if __name__ == '__main__':
    app = QApplication(sys.argv)
    app.setStyleSheet(style_sheet)
    window = FoodOrderGUI()
    sys.exit(app.exec_())
```

When finished, your GUI should look similar to the one in Figure 6-2.

Explanation

Let's first import the modules we need for this project. Next, the properties for the widgets in this application are prepared in the style_sheet variable. We will get to how this works shortly.

Create the structure for the tabs and layout for the main window in setupTabsAndLayout(). Set up instances of the QTabWidget and QWidget objects that will be used for the pages of the tabs. The two tabs are the pizza_tab, to display choices for building your own pizza, and the wings_tab, to show choices for wings flavors.

Some of the widgets in this GUI are given an ID selector using the setObjectName() method. For example, pizza_tab is given the Tabs ID selector. This name will be used in the style_sheet to differentiate this widget from other QWidget objects with a different style.

```
        self.pizza_tab = QWidget()
        self.pizza_tab.setObjectName("Tabs")
```

The side_widget is used to give feedback to users of their choices and can be seen even if the user switches tabs. All of the child widgets for side_widget are then arranged in a nested layout and added to the main QHBoxLayout.

The pizzaTab() method creates and arranges the child widgets for the first tab, pizza_tab. The top of the first page gives users information about the purpose of the tab using images and text. The wingsTab() method is set up in a similar manner.

QRadioButton widgets are grouped together using group boxes. This allows each group to have a title. The QGroupBox class does provide exclusivity to radio buttons, but to get the type of functionality to find out which buttons are checked and return their text values, the QRadioButton objects are also grouped using QGroupButton. Refer to Chapter 4 for more information about QButtonGroup. While only one radio button can be selected in the Crust group, users need to be able to select more than one topping. This is achieved by setting the exclusivity of the toppings_group to False.

```
        self.toppings_group.setExclusive(False)
```

If users press the add_to_order_button on either page, the text from the selected radio buttons is displayed in the side_widget. A Python try-except clause is used to ensure that the user has selected radio buttons.

Applying the Style Sheet

If a style sheet is not applied to the food ordering GUI, then it will use your system's native settings to style the application. Figure 6-5 shows what this looks like on MacOS.
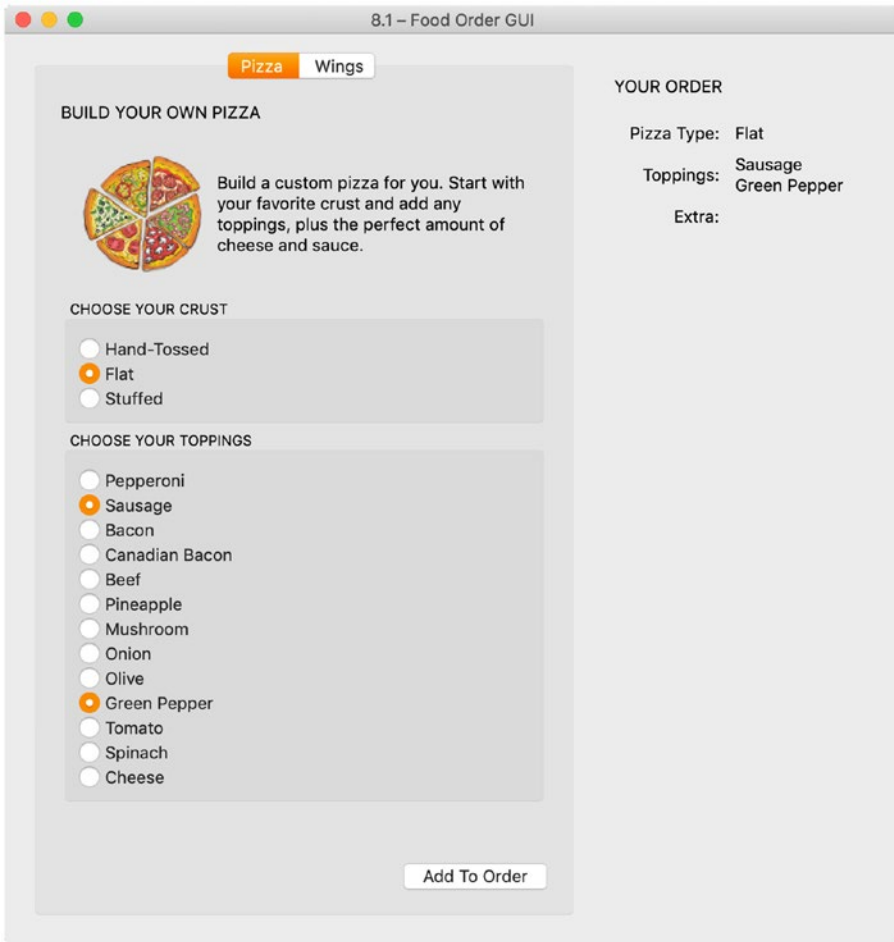
Figure 6-5. *The food ordering GUI before the style sheet is applied*

In the beginning of the program, you will notice the style_sheet variable that holds all of the different style specifications for the different widgets.

To apply a general style to all widgets of one type, you only need to specify the class. For example, the following code gives all QWidget objects a red background:

```
QWidget{
    background-color: #C92108;
}
```

But if a QWidget object has a specified ID selector such as Tabs, then it will get a tan background and rounded corners.

```
QWidget#Tabs{
    background-color: #FCEBCD;
    border-radius: 4px
}
```

Other widget's properties are set up in a similar manner. The style sheet is applied to the entire application by calling setStyleSheet() on the QApplication object.

```
app.setStyleSheet(style_sheet)
```

The final GUI with customized colors, borders, and fonts can be seen in Figure 6-2.

Event Handling in PyQt

The concept of signals and slots in PyQt was briefly introduced in Chapter 3. Event handling in PyQt uses signals and slots to communicate between objects. Signals are typically generated by a user's actions, and slots are methods that are executed in response to the signal. For example, when a QPushButton is pushed, it emits a clicked() signal. This signal could be connected to the PyQt slot close() so that a user can quit the application when the button is pressed.

The clicked() signal is but one of many predefined Qt signals. The type of signals that can be emitted differs according to the widget class. PyQt delivers events to widgets by calling specific, predefined handler functions. These can range from functions related to window operations, such as show() or close(), to GUI appearances with setStyleSheet(), to mouse press and release events, and more.

The way in which event handlers deal with events can also be reimplemented. You saw an example of this back in Chapter 3 when the closeEvent() function was modified to display dialog boxes before closing the application.

The following example, Listing 6-3, shows a very simple example of how to reimplement the keyPressEvent() function.

Listing 6-3. Code to demonstrate how to modify event handlers

```
# close_event.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow
from PyQt5.QtCore import Qt
```

```python
class Example(QMainWindow):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 300, 200)
        self.setWindowTitle('Event Handling Example')
        self.show()

    def keyPressEvent(self, event):
        if event.key() == Qt.Key_Escape:
            print("Application closed.")
            self.close()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = Example()
    sys.exit(app.exec_())
```

Explanation

Whenever a user presses a key on the keyboard, it sends a signal to the computer. If you want to give certain keys abilities, then you will need to use the keyPressEvent().

The keyPressEvent() function checks for events, which in this case are the signals being sent from keys. If the key pressed is the Escape key, then the application calls the close() function to quit the application.

Of course, you can check for any type of key with the keyPressEvent() and cause it to perform any number of actions.

Creating Custom Signals

We have taken a look at some of PyQt's predefined signals and slots. For many of the projects in previous chapters, we have also created custom slots to handle the signals emitted from widgets.

Now let's see how we can create a custom signal using `pyqtSignal` to change a widget's style sheet in Listing 6-4.

Listing 6-4. Creating a custom signal to change the background color of a QLabel widget

```
# color_event.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel
from PyQt5.QtCore import Qt, pyqtSignal, QObject

class SendSignal(QObject):
    """
    Define a signal change_style that takes no arguments.
    """
    change_style = pyqtSignal()

class Example(QMainWindow):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 300, 200)
        self.setWindowTitle('Create Custom Signals')

        self.setupLabel()

        self.show()
```

```python
def setupLabel(self):
    """
    Create label and connect custom signal to slot.
    """
    self.index = 0 # index of items in list
    self.direction = ""

    self.colors_list = ["red", "orange", "yellow", "green", "blue",
    "purple"]

    self.label = QLabel()
    self.label.setStyleSheet("background-color: {}".format(self.colors_
    list[self.index]))
    self.setCentralWidget(self.label)

    # Create instance of SendSignal class, and
    # connect change_style signal to a slot.
    self.sig = SendSignal()
    self.sig.change_style.connect(self.changeBackground)

def keyPressEvent(self, event):
    """
    Reimplement how the key press event is handled.
    """
    if (event.key() == Qt.Key_Up):
        self.direction = "up"
        self.sig.change_style.emit()
    elif event.key() == Qt.Key_Down:
        self.direction = "down"
        self.sig.change_style.emit()

def changeBackground(self):
    """
    Change the background of the label widget when a keyPressEvent
    signal is emitted.
    """
```

```
        if self.direction == "up" and self.index < len(self.colors_list) - 1:
            self.index = self.index + 1
            self.label.setStyleSheet("background-color: {}".format(self.
            colors_list[self.index]))
        elif self.direction == "down" and self.index > 0:
            self.index = self.index - 1
            self.label.setStyleSheet("background-color: {}".format(self.
            colors_list[self.index]))
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = Example()
    sys.exit(app.exec_())
```

This example creates a simple GUI with a QLabel widget as the central widget of the main window.

Explanation

The pyqtSignal factory and QObject classes are imported from the QtCore module. The QtCore module and QObject classes provide the mechanics for signals and slots.

The SendSignal class creates a new signal called change_style from the pyqtSignal factory. This signal will be generated whenever the user presses either the up arrow key or the down arrow key. By pressing up or down, the user can change the background color of the QLabel object.

```
    self.sig.change_style.connect(self.changeBackground)
```

When the user presses Key_Up, direction is set equal to "up", and a change_style signal is emitted.

```
    self.sig.change_style.emit()
```

This signal is connected to the changeBackground() slot which updates the color of the label by calling the setStyleSheet() method.

It works in a similar fashion when the down key is pressed.

Summary

In this chapter, we saw how to use Qt Style Sheets to modify the appearance of widgets to better fit the purpose and look of an application. Applying customizations in a consistent and attentive manner can greatly influence the usability of a user interface.

Allowing the user to also have some control over the look of the window can improve the user's experience. This can be done in a number of ways – through the menu or toolbar, using a context menu, or even through simple presses of keys on the keyboard.

Chapter 7 will introduce Qt Designer, a tool that will make the process for designing GUIs much simpler.

Creating GUIs with Qt Designer

The previous chapters have focused on learning how to manually code GUIs using PyQt. This was done intentionally so that you could have a better fundamental understanding of the code and processes used to create simple applications. Chapters 2 and 3 showed you how to create your own GUI from scratch. In Chapter 4, you learned about layouts and how to arrange widgets by coding them yourself. You saw how to create applications with menus and toolbars in Chapter 5 and how to style their look in Chapter 6.

While setting up and arranging GUIs yourself gives you more control over the design process, not everyone will need or want to take the time to do so. Fortunately, Qt provides a great application for setting up the layouts and designing main windows, widgets, or dialogs. **Qt Designer** is a graphical interface filled with Qt widgets and other tools used for building GUIs. Using the Qt Designer application's drag and drop interface, you are able to create and customize your own dialogs, windows, and widgets.

The widgets and other applications you create using Qt Designer integrate with programmed code, using Qt's signals and slots mechanism, so that you can easily assign behavior to widgets and graphical elements. This means that rather than focusing most of your time on layout and design, you can get into coding the functionality of an application much faster.

In Chapter 7, you will

- Find out about the Qt Designer user interface

- Create an application in Qt Designer, including how to set layouts, edit object properties, connect signals and slots, and generate Python code

- Learn about the QFrame class for containing other widgets

- Be introduced to a couple new Qt classes including QPalette and QIntValidator

---

Tip   For references or more help beyond the scope of this chapter, check out the Qt Documentation for Qt Designer at https://doc.qt.io/qt-5/qtdesigner-manual.html.

Getting Started with Qt Designer

Once you have installed PyQt, the first thing you need to do is to launch the Qt Designer application. After opening Qt Designer, you will see a graphical user interface for creating your own GUIs like the one in Figure 7-1.

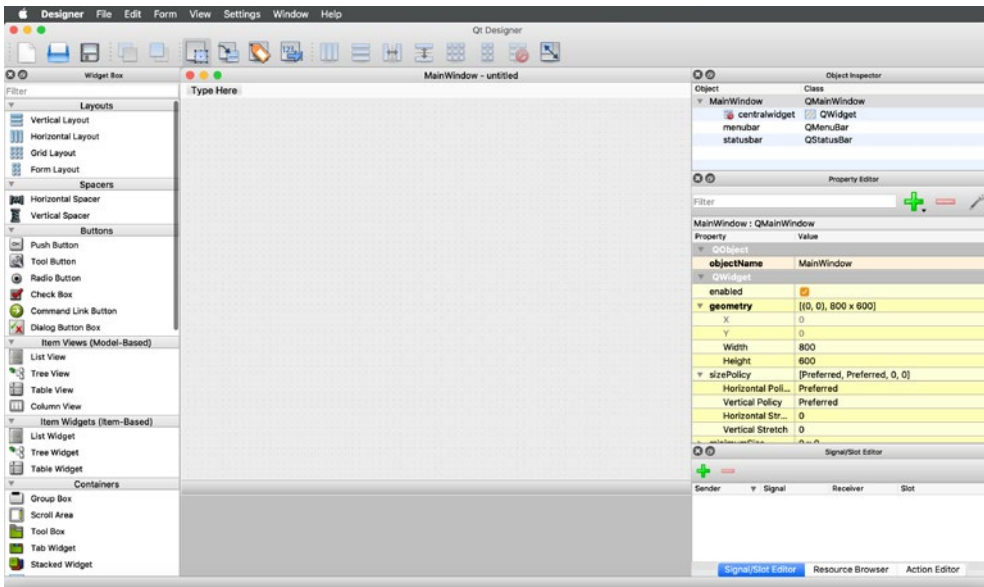**Note**   For more information about downloading and launching Qt Designer for Windows, MacOS, and Linux, refer to Appendix A.

Figure 7-1. *The Qt Designer user interface*

> **Tip**   If you are using a stand-alone version of Qt Designer, then you can
> change the configuration of Qt Designer under `Settings`. Locate the
> `Preferences` menu, and in the dialog box that appears, you can change
> the appearance of Qt Designer from a `Multi Top-Level Windows` user
> interface to one using `Docked Windows`. Otherwise, the multilevel layout is
> only available if you use Qt Creator, the integrated development environment
> (IDE) for working with Qt.

Before you create your first application, let's get to know the different menus, tools, and modes that are displayed in the main window in Figure 7-1.

Exploring Qt Designer's User Interface

When you first open up Qt Designer, you will notice a dialog in the center of the window with the title `New Form`. This dialog can be seen in Figure 7-2. From here you can select a template for creating a main window, a widget, or different kinds of dialog boxes. You can also choose what kinds of widgets to add to your project's layout. Once you have selected a template and the application's size, an empty window, also known as a **form**, will appear for you to modify.
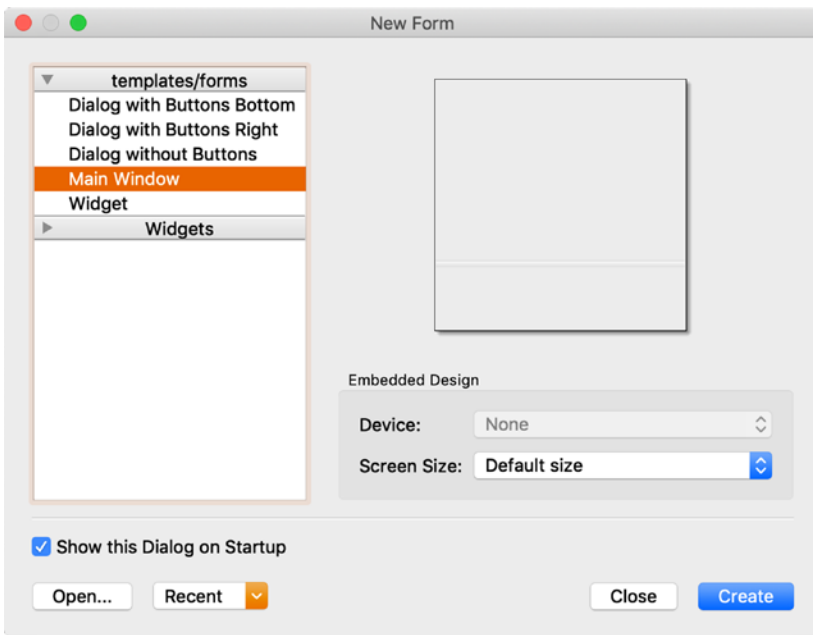
Figure 7-2. *The New Form dialog box for selecting what type of application to build*

At the top of the main window in Figure 7-1, you will notice Qt Designer's menubar and toolbar for managing and editing your GUI. On the left side of the main window is the `Widget Box` dock widget, shown in Figure 7-3, which provides an organized list of layouts and widgets that can be dragged and dropped onto the required locations of your GUI. Other features for tinkering with the form can be accessed by right-clicking and opening up various context menus.

Another very useful dock widget is the `Property Editor` displayed in Figure 7-4. The properties of windows, widgets, and layouts such as an object's name, size constraints, status tips, and more can all be altered using the `Property Editor`. Each widget you add to a form will have its own set of properties as well as ones that the widget inherits from other classes. To select a specific widget, you can either click the object in the form or the widget's name in the `Object Inspector` dock widget.

The `Object Inspector` allows you to view all of the objects that are currently being used as well as their hierarchical layout. In Figure 7-5, you can see how the `MainWindow` is listed first, followed by the `centralwidget` and all of its widgets. If your form also has a menu or toolbar, then they will also be listed in the `Object Inspector` along with their corresponding actions.

Note   The main layout for your GUI is not displayed in the `Object Inspector`. A broken layout icon (a red circle with a slash) is displayed on the central widget or on containers if no layout has been assigned to them.
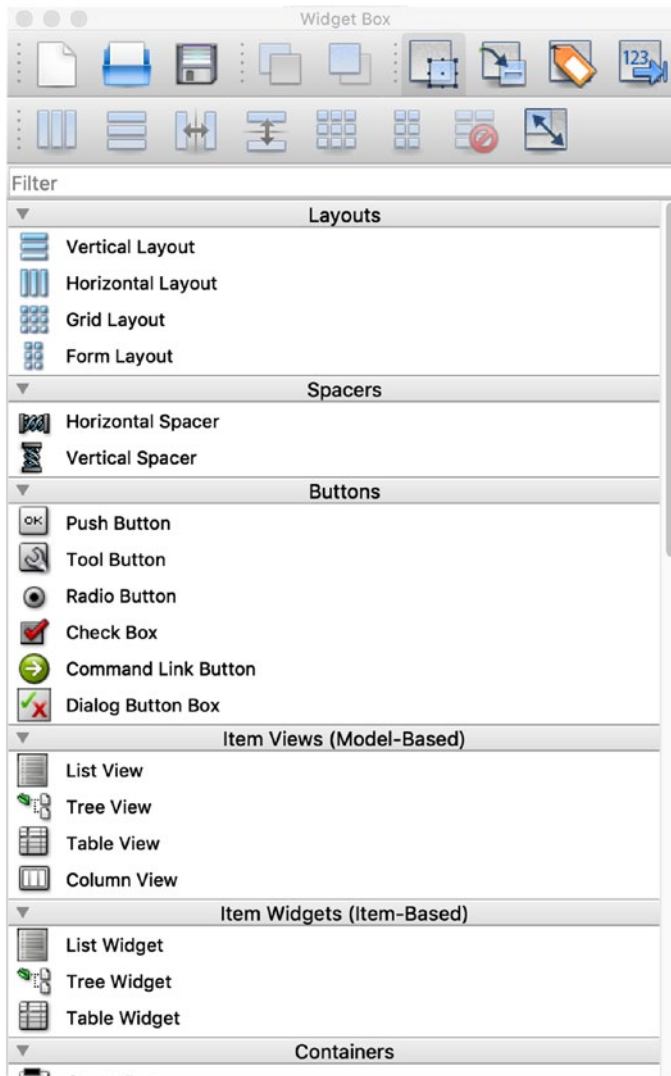

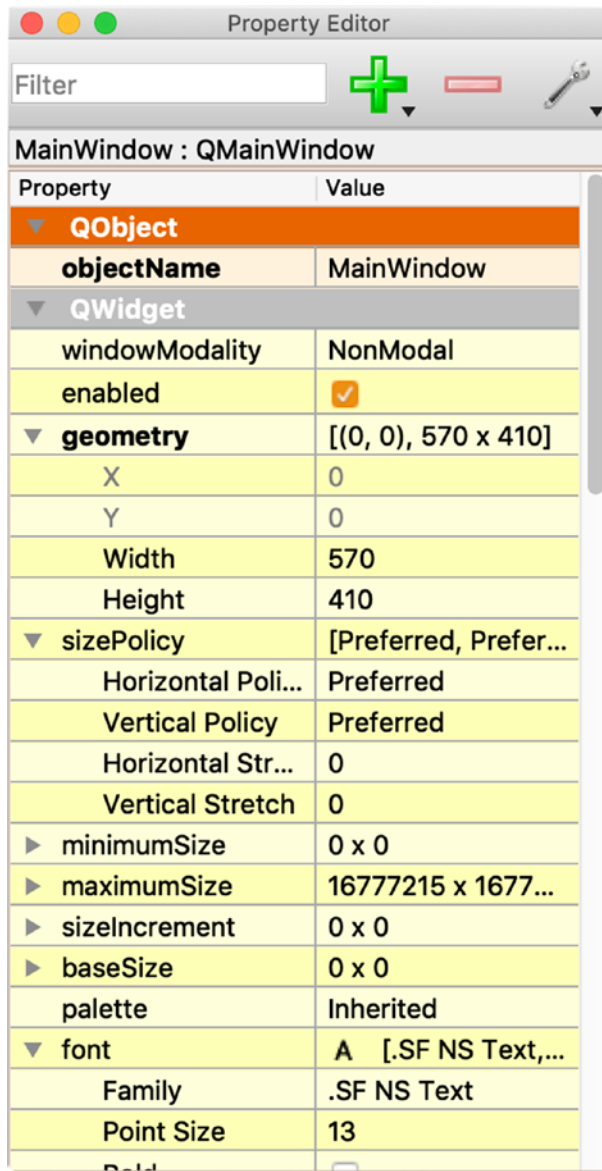*Figure 7-3.* *The Widget Box dock widget for selecting layouts and widgets*

Figure 7-4. *The Property Editor dock widget is used for setting the attributes of widgets*
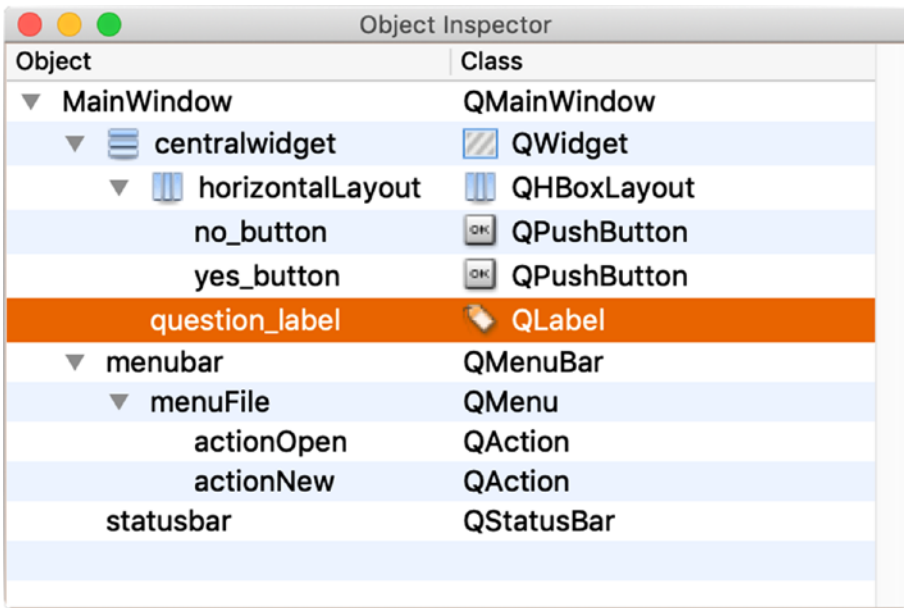
Figure 7-5. *The Object Inspector displays the widget, layout, and menu objects*

In Qt Designer, it is also possible to create, edit, and delete signals and slots between objects using the Signal/Slot Editor. You should be aware that although you can connect signals and slots, you will not always be able to completely configure your widgets and will sometimes need to complete that yourself later in the code. The Signal/Slot Editor can be seen in Figure 7-6. Qt Designer also provides an editing mode for connecting widgets.
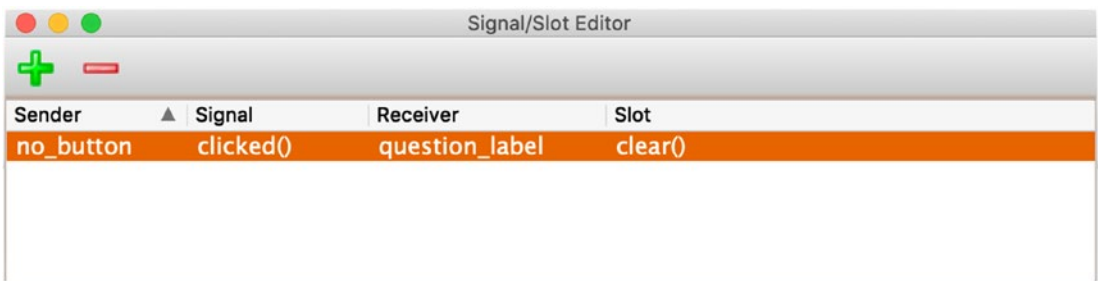

Figure 7-6. *The Signal/Slot Editor for connecting the signals and slots of objects*

Items in a menu, submenu, or a toolbar are assigned commands by using actions. These actions can then be given a shortcut key, made checkable, and more. The Action Editor seen in Figure 7-7 gives you access to working with actions. For more information about assigning actions, refer to Chapter 5.
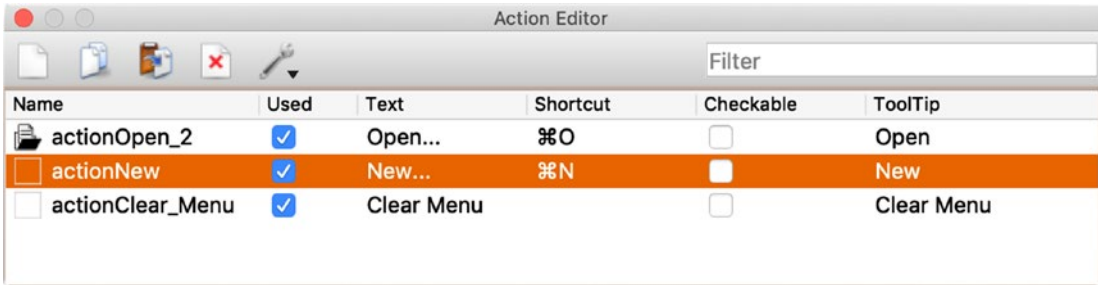

Figure 7-7. *The Action Editor is used to manage the actions of menu items*

Finally, there is the Resource Browser which allows you specify and manage resources you need to use in your application. These resources can include images and icons. The Resource Browser dock widget can be seen in Figure 7-8.

If you need to add resources, you first need to create a new resource file. To do so, click the pencil in the top-left corner of the Resource Browser dock widget. This will open an Edit Resources dialog similar to the one in Figure 7-9. Next, click the Create New Resource button, navigate to the correct directory, and enter a name for the resource file. The file will be saved with a .qrc file extension, which stands for Qt Resource Collection and contains a list of all the resources used in your program. From here, create a prefix for managing the types of resources and begin adding files such as images and icons. When you are finished, click the OK button and the files will be added to the Resource Browser.

To access files in code found in the resource file, append ":/" to the beginning of the file's location. For example, to use the new_file icon

```
self.label.setPixmap(QtGui.QPixmap(":/icons/images/new_file.png"))
```
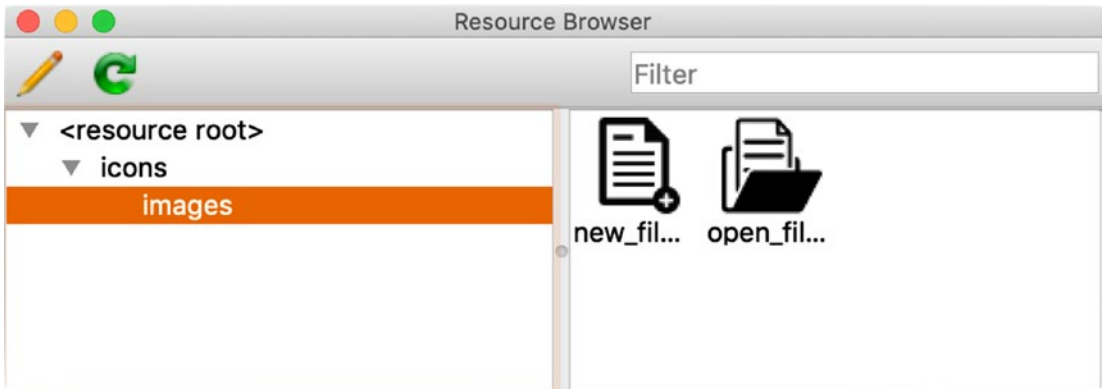
Figure 7-8. *The Resource Browser for working with resources such as images and icons*
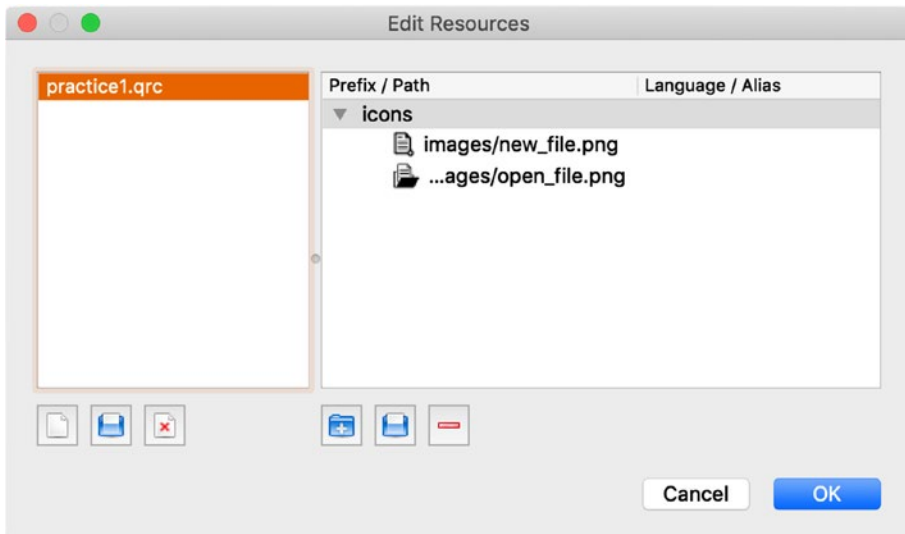
Figure 7-9. *The Edit Resources dialog*

Qt Designer's Editing Modes

In Qt Designer there are four different editing modes that can be accessed either in the Edit menu or from Qt Designer's toolbar. Take a look at Figure 7-10 to help you locate the widgets in the toolbar.

1. Edit Widgets – Widgets can be dragged and dropped to a form, layouts can be applied, and objects can be edited both on the form and in the Property Editor. This is the default mode.

173

2. `Edit Signals/Slots` – Connect signals and slots for widgets and
   layouts. To create connections, click an object and drag the cursor
   toward an object that will receive the signal. Items that can be
   connected will be highlighted as the mouse cursor moves over
   them. To create the connection, release the mouse button once
   a line with an arrow connects the two objects. Then configure
   the signals and slots. Use in conjunction with the `Signal/Slots`
   `Editor` dock widget to edit connections.

3. `Edit Buddies` – Connect QLabel widgets with shortcuts to
   input widgets such as QLineEdit or QTextEdit. The input widget
   becomes the QLabel's "buddy." When the user enters the label's
   shortcut key, the focus moves to the input widget.

4. `Edit Tab Order` – Set the order in which widgets receive focus
   when the `tab` key is pressed. This allows the user to navigate
   through the different widgets to make your application easier to use.

Figure 7-10. *Qt Designer's Editing Modes (outlined in red). (From left to right)*
Edit Widgets, Edit Signals/Slots, Edit Buddies, Edit Tab Order

Creating an Application in Qt Designer

When you are creating your GUI's windows and widgets, you will probably continue to
make slight adjustments to your application before it is finished. Fortunately, there are a
few steps you can follow to simplify the building process.

1. Select a form – In the `New Form` dialog (shown in Figure 7-2),
   choose from one of the available templates, Main Window,
   Widget, or a type of dialog. You can also add and preview widgets
   to include in your GUI.

2. Arrange objects on the form – Use Qt Designer's drag and drop
   mechanics to place widgets on the form. Then assign layouts to
   containers and the main window.

3. Edit the properties of objects – Click the objects in the form and edit their features in the `Property Editor` dock widget.

4. Connect signals and slots – Use the `Signal/Slots` Editing mode to link signals to slots.

5. Preview your GUI – Examine the form before saving it as a UI file with the `.ui` extension.

6. Create and edit Python code – Utilize the `pyuic` compiler to convert the UI file to readable and editable Python code.

The following project will cover these steps in addition to many of the basic concepts for creating GUIs using Qt Designer.

Project 7.1 – Keypad GUI

To get you started using Qt Designer, the project in this chapter is a simple one – a keypad GUI. A keypad is a set of buttons with digits, symbols, or letters used as an input device for passcodes, telephone numbers, and more. They can be found on a number of devices such as calculators, cell phones, and locks. Figure 7-11 shows the keypad GUI you will create in this project.
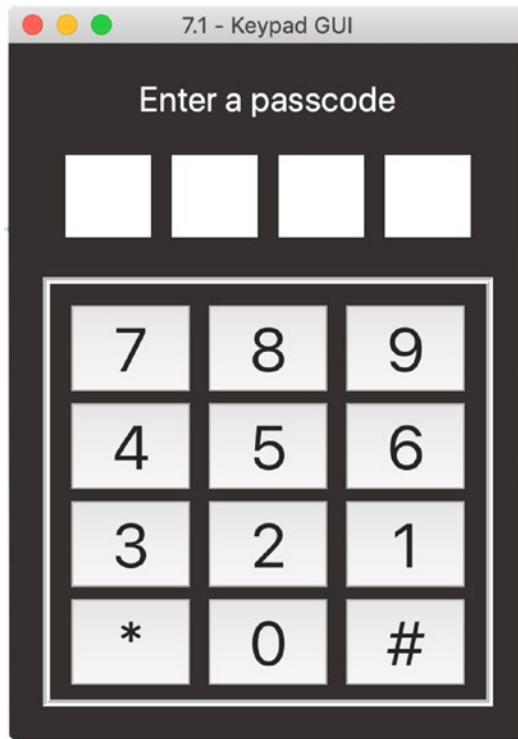
Figure 7-11. *Keypad GUI*

Keypad GUI Solution

The keypad application is comprised of two Python files, keypad_gui.py and keypad_
main.py. The keypad_gui.py contains all of the Python code generated from the UI file
created using Qt Designer. In order to use that code, we then create a customized class in
a separate file, keypad_main.py, to import and set up the GUI.

The keypad GUI consists of four QLineEdit widgets to input only numeric values,
twelve QPushButton widgets, and a single QLabel to display information about how to
use the interface. The asterisk button allows users to clear the current input and the hash
button is for confirming the user's four-digit input.

This project introduces the QFrame container for organizing Qt widgets. The
program also utilizes nested layouts to arrange the various widgets and containers.

> **Note**   The following Python code in Listing 7-1 is produced from the UI file using
> `pyuic`. It has not been altered. To help you understand the code, Listing 7-1 is
> broken into parts with annotations.

Listing 7-1. Code for keypad created from keypad.ui

```python
# keypad_gui.py
# Import necessary modules
from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_Keypad(object):
    def setupUi(self, Keypad):
        Keypad.setObjectName("Keypad")
        Keypad.resize(302, 406)
```

Import modules from PyQt5 and create a class that inherits from QWidget, Keypad.
The member function setupUi() of the class Ui_Keypad is used to build a widget tree
on the parent widget, Keypad. A **widget tree** is used to represent the organization of
widgets in a UI. So the setupUi() method composes the UI based upon the widgets and
connections we used to create it along with the parameters it inherits from its parent
widget.

```python
palette = QtGui.QPalette()
brush = QtGui.QBrush(QtGui.QColor(255, 255, 255))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Active, QtGui.QPalette.Base, brush)
brush = QtGui.QBrush(QtGui.QColor(52, 48, 47))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Active, QtGui.QPalette.Window, brush)
brush = QtGui.QBrush(QtGui.QColor(255, 255, 255))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.Base, brush)
brush = QtGui.QBrush(QtGui.QColor(52, 48, 47))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.Window, brush)
brush = QtGui.QBrush(QtGui.QColor(52, 48, 47))
```

```
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Disabled, QtGui.QPalette.Base, brush)
brush = QtGui.QBrush(QtGui.QColor(52, 48, 47))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Disabled, QtGui.QPalette.Window, brush)
Keypad.setPalette(palette)
```

Every widget in Qt has a palette which contains information about how they will be drawn in the window. The **QPalette** class contains the color groups for each widget during one of three possible states – Active, Inactive, or Disabled. The preceding code changes the color of the main window to dark gray. How to change the settings in the palette will be introduced in the "Explanation" section of Project 7.1. You can also create style sheets in Qt Designer.

```
self.verticalLayout = QtWidgets.QVBoxLayout(Keypad)
self.verticalLayout.setObjectName("verticalLayout")
```

Create the vertical layout that will be used for the main window.

```
self.label = QtWidgets.QLabel(Keypad)
palette = QtGui.QPalette()
brush = QtGui.QBrush(QtGui.QColor(255, 255, 255))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Active, QtGui.QPalette.WindowText, brush)
brush = QtGui.QBrush(QtGui.QColor(255, 255, 255))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.WindowText, brush)
brush = QtGui.QBrush(QtGui.QColor(127, 127, 127))
brush.setStyle(QtCore.Qt.SolidPattern)
palette.setBrush(QtGui.QPalette.Disabled, QtGui.QPalette.WindowText, brush)
self.label.setPalette(palette)
font = QtGui.QFont()
font.setPointSize(20)
self.label.setFont(font)
self.label.setAlignment(QtCore.Qt.AlignCenter)
self.label.setObjectName("label")
self.verticalLayout.addWidget(self.label)
```

Create the QLabel object, modify its palette settings so that the color of the font is light gray, and add the label to the vertical layout.

```
self.frame = QtWidgets.QFrame(Keypad)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Preferred,
QtWidgets.QSizePolicy.Preferred)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(1)
sizePolicy.setHeightForWidth(self.frame.sizePolicy().hasHeightForWidth())
self.frame.setSizePolicy(sizePolicy)
self.frame.setFrameShape(QtWidgets.QFrame.NoFrame)
self.frame.setFrameShadow(QtWidgets.QFrame.Plain)
self.frame.setLineWidth(0)
self.frame.setObjectName("frame")
self.horizontalLayout = QtWidgets.QHBoxLayout(self.frame)
self.horizontalLayout.setObjectName("horizontalLayout")
```

The first `QFrame` container, `frame`, will hold four `QLineEdit` widgets and use a horizontal layout. You can adjust the style of a frame object using its `setFrameShape()`, `setFrameShadow()`, and other methods.

```
self.line_edit1 = QtWidgets.QLineEdit(self.frame)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.line_edit1.sizePolicy().
hasHeightForWidth())
self.line_edit1.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(30)
self.line_edit1.setFont(font)
self.line_edit1.setAlignment(QtCore.Qt.AlignCenter)
self.line_edit1.setObjectName("line_edit1")
self.horizontalLayout.addWidget(self.line_edit1)
self.line_edit2 = QtWidgets.QLineEdit(self.frame)
```

```
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.line_edit2.sizePolicy().
hasHeightForWidth())
self.line_edit2.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(30)
self.line_edit2.setFont(font)
self.line_edit2.setAlignment(QtCore.Qt.AlignCenter)
self.line_edit2.setObjectName("line_edit2")
self.horizontalLayout.addWidget(self.line_edit2)
self.line_edit3 = QtWidgets.QLineEdit(self.frame)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.line_edit3.sizePolicy().
hasHeightForWidth())
self.line_edit3.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(30)
self.line_edit3.setFont(font)
self.line_edit3.setAlignment(QtCore.Qt.AlignCenter)
self.line_edit3.setObjectName("line_edit3")
self.horizontalLayout.addWidget(self.line_edit3)
self.line_edit4 = QtWidgets.QLineEdit(self.frame)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.line_edit4.sizePolicy().
hasHeightForWidth())
self.line_edit4.setSizePolicy(sizePolicy)
font = QtGui.QFont()
```

```
font.setPointSize(30)
self.line_edit4.setFont(font)
self.line_edit4.setAlignment(QtCore.Qt.AlignCenter)
self.line_edit4.setObjectName("line_edit4")
self.horizontalLayout.addWidget(self.line_edit4)
self.verticalLayout.addWidget(self.frame)
```

Each of the four line edit widgets has size policies that allow them to stretch if the window resizes in both the vertical and horizontal directions by using QSizePolicy. Expanding. They are then arranged in the horizontalLayout of the frame container. The frame object is then added to the verticalLayout of the main window.

```
self.frame_2 = QtWidgets.QFrame(Keypad)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Preferred,
QtWidgets.QSizePolicy.Preferred)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(2)
sizePolicy.setHeightForWidth(self.frame_2.sizePolicy().hasHeightForWidth())
self.frame_2.setSizePolicy(sizePolicy)
self.frame_2.setFrameShape(QtWidgets.QFrame.Box)
self.frame_2.setFrameShadow(QtWidgets.QFrame.Sunken)
self.frame_2.setLineWidth(2)
self.frame_2.setObjectName("frame_2")
self.gridLayout = QtWidgets.QGridLayout(self.frame_2)
self.gridLayout.setObjectName("gridLayout")
```

Instantiate the second frame container, and set the size policy and style attributes. The layout inside frame_2 houses the twelve buttons and uses a grid layout.

```
self.button_7 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_7.sizePolicy().
hasHeightForWidth())
self.button_7.setSizePolicy(sizePolicy)
```

```
font = QtGui.QFont()
font.setPointSize(36)
self.button_7.setFont(font)
self.button_7.setObjectName("button_7")
self.gridLayout.addWidget(self.button_7, 0, 0, 1, 1)
self.button_8 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.MinimumExpanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_8.sizePolicy().
hasHeightForWidth())
self.button_8.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_8.setFont(font)
self.button_8.setObjectName("button_8")
self.gridLayout.addWidget(self.button_8, 0, 1, 1, 1)
self.button_9 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.MinimumExpanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_9.sizePolicy().
hasHeightForWidth())
self.button_9.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_9.setFont(font)
self.button_9.setObjectName("button_9")
self.gridLayout.addWidget(self.button_9, 0, 2, 1, 1)
self.button_4 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.MinimumExpanding)
sizePolicy.setHorizontalStretch(0)
```

```
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_4.sizePolicy().
hasHeightForWidth())
self.button_4.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_4.setFont(font)
self.button_4.setObjectName("button_4")
self.gridLayout.addWidget(self.button_4, 1, 0, 1, 1)
self.button_5 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.MinimumExpanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_5.sizePolicy().
hasHeightForWidth())
self.button_5.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_5.setFont(font)
self.button_5.setObjectName("button_5")
self.gridLayout.addWidget(self.button_5, 1, 1, 1, 1)
self.button_6 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.MinimumExpanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_6.sizePolicy().
hasHeightForWidth())
self.button_6.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_6.setFont(font)
self.button_6.setObjectName("button_6")
self.gridLayout.addWidget(self.button_6, 1, 2, 1, 1)
```

```
self.button_3 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_3.sizePolicy().
hasHeightForWidth())
self.button_3.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_3.setFont(font)
self.button_3.setObjectName("button_3")
self.gridLayout.addWidget(self.button_3, 2, 0, 1, 1)
self.button_2 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_2.sizePolicy().
hasHeightForWidth())
self.button_2.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_2.setFont(font)
self.button_2.setObjectName("button_2")
self.gridLayout.addWidget(self.button_2, 2, 1, 1, 1)
self.button_1 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_1.sizePolicy().
hasHeightForWidth())
self.button_1.setSizePolicy(sizePolicy)
font = QtGui.QFont()
```

```
font.setPointSize(36)
self.button_1.setFont(font)
self.button_1.setObjectName("button_1")
self.gridLayout.addWidget(self.button_1, 2, 2, 1, 1)
self.button_star = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_star.sizePolicy().
hasHeightForWidth())
self.button_star.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_star.setFont(font)
self.button_star.setObjectName("button_star")
self.gridLayout.addWidget(self.button_star, 3, 0, 1, 1)
self.button_0 = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.Expanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
sizePolicy.setHeightForWidth(self.button_0.sizePolicy().
hasHeightForWidth())
self.button_0.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_0.setFont(font)
self.button_0.setObjectName("button_0")
self.gridLayout.addWidget(self.button_0, 3, 1, 1, 1)
self.button_hash = QtWidgets.QPushButton(self.frame_2)
sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.Minimum,
QtWidgets.QSizePolicy.MinimumExpanding)
sizePolicy.setHorizontalStretch(0)
sizePolicy.setVerticalStretch(0)
```

```
sizePolicy.setHeightForWidth(self.button_hash.sizePolicy().
hasHeightForWidth())
self.button_hash.setSizePolicy(sizePolicy)
font = QtGui.QFont()
font.setPointSize(36)
self.button_hash.setFont(font)
self.button_hash.setObjectName("button_hash")
self.gridLayout.addWidget(self.button_hash, 3, 2, 1, 1)
self.verticalLayout.addWidget(self.frame_2)
```

The twelve QPushButton widgets are created, and their properties, such as their object names and font sizes, are adjusted. Every button is then added to the grid layout of frame_2 which is then added to the vertical layout of the main window.

```
self.retranslateUi(Keypad)
self.button_star.clicked.connect(self.line_edit1.clear)
self.button_star.clicked.connect(self.line_edit2.clear)
self.button_star.clicked.connect(self.line_edit3.clear)
self.button_star.clicked.connect(self.line_edit4.clear)
QtCore.QMetaObject.connectSlotsByName(Keypad)
```

The retranslateUi() method handles how to display text in the GUI in the situation where a different language is used. In the keypad, the user is given a way to delete their input and try again. When the button_star is clicked, it sends a signal to clear the text in all four QLineEdit widgets. This could be handled a different way, but for this example, this is used as an example to show how to connect signals to slots in Qt Designer.

```
def retranslateUi(self, Keypad):
_translate = QtCore.QCoreApplication.translate
Keypad.setWindowTitle(_translate("Keypad", "7.1 - Keypad GUI"))
self.label.setText(_translate("Keypad", "Enter a passcode"))
self.button_7.setText(_translate("Keypad", "7"))
self.button_8.setText(_translate("Keypad", "8"))
self.button_9.setText(_translate("Keypad", "9"))
self.button_4.setText(_translate("Keypad", "4"))
self.button_5.setText(_translate("Keypad", "5"))
self.button_6.setText(_translate("Keypad", "6"))
self.button_3.setText(_translate("Keypad", "3"))
```

```
self.button_2.setText(_translate("Keypad", "2"))
self.button_1.setText(_translate("Keypad", "1"))
self.button_star.setText(_translate("Keypad", "*"))
self.button_0.setText(_translate("Keypad", "0"))
self.button_hash.setText(_translate("Keypad", "#"))
```

The following code in Listing 7-2 creates the class that inherits from Ui_Keypad and
sets up the GUI application.

Listing 7-2. Code for the keypad GUI created from keypad.ui

```
# keypad_main.py
# Import necessary modules
import sys
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import QMessageBox
from PyQt5.QtGui import QIntValidator
from keypad_gui import Ui_Keypad

class KeypadGUI(QtWidgets.QWidget):
    def __init__(self):
        super(KeypadGUI, self).__init__()
        self.ui = Ui_Keypad()
        self.ui.setupUi(self)

        self.initializeUI()

        self.show()

    def initializeUI(self):
        # Update other line_edit features
        self.ui.line_edit1.setMaxLength(1) # Set the max number of
        characters allowed
        self.ui.line_edit1.setValidator(QIntValidator(0, 9)) # User can
        only enter ints from 0-9
        self.ui.line_edit1.setFocusPolicy(QtCore.Qt.NoFocus) # Widget does
        not except focus
```

```
self.ui.line_edit2.setMaxLength(1)
self.ui.line_edit2.setValidator(QIntValidator(0, 9))
self.ui.line_edit2.setFocusPolicy(QtCore.Qt.NoFocus)

self.ui.line_edit3.setMaxLength(1)
self.ui.line_edit3.setValidator(QIntValidator(0, 9))
self.ui.line_edit3.setFocusPolicy(QtCore.Qt.NoFocus)

self.ui.line_edit4.setMaxLength(1)
self.ui.line_edit4.setValidator(QIntValidator(0, 9))
self.ui.line_edit4.setFocusPolicy(QtCore.Qt.NoFocus)

# 4-digit passcode
self.passcode = 8618

#### Add signal/slot connections for buttons ####
self.ui.button_0.clicked.connect(lambda: self.numberClicked(self.
ui.button_0.text()))
self.ui.button_1.clicked.connect(lambda: self.numberClicked(self.
ui.button_1.text()))
self.ui.button_2.clicked.connect(lambda: self.numberClicked(self.
ui.button_2.text()))
self.ui.button_3.clicked.connect(lambda: self.numberClicked(self.
ui.button_3.text()))
self.ui.button_4.clicked.connect(lambda: self.numberClicked(self.
ui.button_4.text()))
self.ui.button_5.clicked.connect(lambda: self.numberClicked(self.
ui.button_5.text()))
self.ui.button_6.clicked.connect(lambda: self.numberClicked(self.
ui.button_6.text()))
self.ui.button_7.clicked.connect(lambda: self.numberClicked(self.
ui.button_7.text()))
self.ui.button_8.clicked.connect(lambda: self.numberClicked(self.
ui.button_8.text()))
self.ui.button_9.clicked.connect(lambda: self.numberClicked(self.
ui.button_9.text()))
self.ui.button_hash.clicked.connect(self.checkPasscode)
```

```python
def numberClicked(self, text_value):
    """
    When a button with a digit is pressed, check if the text for
    QLineEdit widgets
    are empty. If empty, set the focus to the correct widget and enter
    text value.
    """
    if self.ui.line_edit1.text() == "":
        self.ui.line_edit1.setFocus()
        self.ui.line_edit1.setText(text_value)
        self.ui.line_edit1.repaint()
    elif (self.ui.line_edit1.text() != "") and (self.ui.line_edit2.
    text() == ""):
        self.ui.line_edit2.setFocus()
        self.ui.line_edit2.setText(text_value)
        self.ui.line_edit2.repaint()
    elif (self.ui.line_edit1.text() != "") and (self.ui.line_edit2.
    text() != "") \
        and (self.ui.line_edit3.text() == ""):
        self.ui.line_edit3.setFocus()
        self.ui.line_edit3.setText(text_value)
        self.ui.line_edit3.repaint()
    elif (self.ui.line_edit1.text() != "") and (self.ui.line_edit2.
    text() != "") \
        and (self.ui.line_edit3.text() != "") and (self.ui.line_edit4.
        text() == ""):
        self.ui.line_edit4.setFocus()
        self.ui.line_edit4.setText(text_value)
        self.ui.line_edit4.repaint()

def checkPasscode(self):
    """
    Concatenate the text values from the 4 QLineEdit widgets, and
    check to see if the passcode entered by user matches the existing
    passcode.
    """
```

```
        entered_passcode = self.ui.line_edit1.text() + self.ui.line_edit2.
        text() + \
            self.ui.line_edit3.text() +  self.ui.line_edit4.text()
        if len(entered_passcode) == 4 and int(entered_passcode) == self.
        passcode:
            QMessageBox.information(self, "Valid Passcode!", "Valid
            Passcode!", QMessageBox.Ok, QMessageBox.Ok)
            self.close()
        else:
            QMessageBox.warning(self, "Error Message", "Invalid Passcode.",
            QMessageBox.Close, QMessageBox.Close)
            self.ui.line_edit1.clear()
            self.ui.line_edit2.clear()
            self.ui.line_edit3.clear()
            self.ui.line_edit4.clear()
            self.ui.line_edit1.setFocus()

if __name__ == "__main__":
    app = QtWidgets.QApplication(sys.argv)
    Keypad = KeypadGUI()
    sys.exit(app.exec_())
```

When you run the code, your GUI should look similar to Figure 7-11.

Explanation

In order to utilize the Ui_Keypad class that was created using Qt Designer, we create a new Python file, keypad_main.py. The KeypadGUI class created in keypad_main.py will inherit from the Ui_Keypad class.

We begin by importing the modules needed for this project, including the Ui_Keypad class and a new Qt class, QIntValidator. Qt provides a few classes that can be used to verify the types of input text. QIntValidator will be used to check if the values input into the QLineEdit widgets are integers.

The KeypadGUI class is created using a **single inheritance** approach where it inherits its properties from a single parent class, QWidget. The user interface is set up in the __init__() method in the following lines:

```
self.ui = Ui_Keypad()
self.ui.setupUi(self)
```

In the initializeUI() method, local modifications are made to the QLineEdit widgets. Here the line edit widget's focus policy is set to NoFocus so that users can only enter input in the correct order.

Then we connect the signals and slots for the button widgets. When each button is clicked, it sends a signal that is connected to the numberClicked() slot. Rather than creating a separate method for each button, the lambda function is used to reuse a method for signals. lambda calls the numberClicked() function and passes it a new parameter every time, in this case the specific text from each button.

When a user clicks a button, that button's number needs to appear in the correct line edit widget from left to right. A widget receives focus if its text() value is empty. On MacOS, the repaint() method was used to update the text in the QLineEdit widgets. repaint() is used to force a widget to update itself.

Finally, if the user presses the # button, the method checkPasscode() checks if the user entered the passcode that matches self.passcode. If the input does not match, the line edit widgets are reset. This project could be designed so that the password is read from a file or from a database.

We have taken a look at the user interface class created from Qt Designer and at the file that inherits from it. The following section shows in detail how to create the GUI in Qt Designer.

Select a Form

Begin by opening up Qt Designer. Choose the Widget template from New Form dialog box. We will use the default screen size. Select Create. This opens up a blank GUI window with a grid of dots inside of the Qt Designer interface like in Figure 7-12.
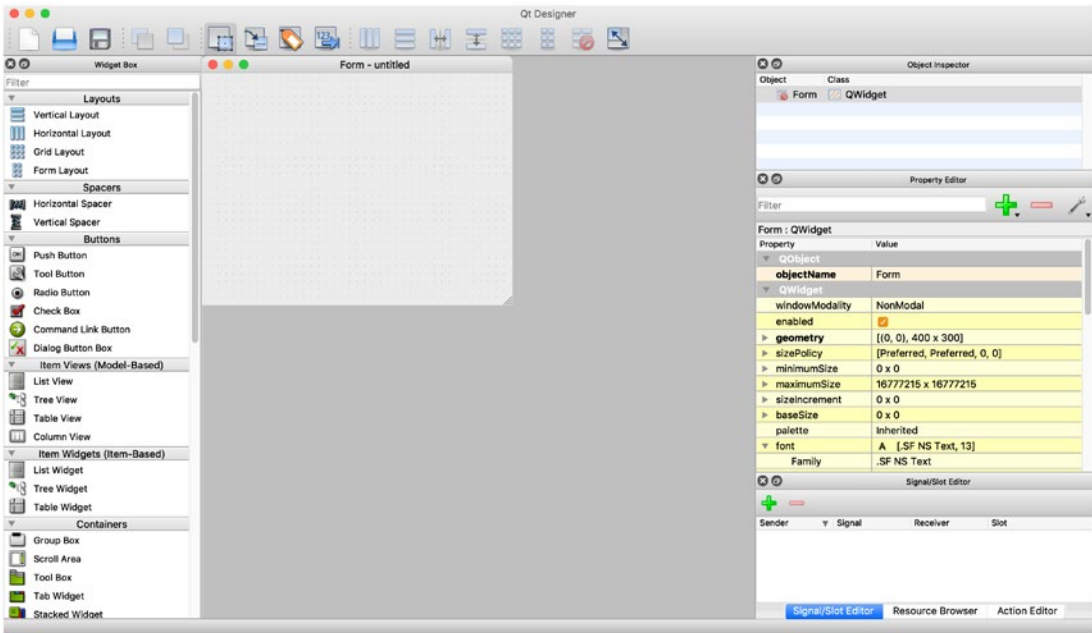
Figure 7-12.   *The Qt Designer interface displaying the toolbar and its different dock widgets for managing GUIs. In the center is the empty Widget form that will be used to create the keypad*

Arrange Objects on the Form

From here you could immediately begin to adjust certain features of the form such as the window size or the background color. Instead, let's begin by adding whatever widgets you may need for your project by dragging and dropping them into the main window from the Widget Box on the left of the window.

Start by selecting a QLabel widget and two QFrame containers and place them on the form like in Figure 7-13. You can resize the frames by clicking them and moving the edges of the frame. Then drag four QLineEdit widgets and arrange them in the top QFrame container. They will overlap, but that will be fixed when you apply layouts to the frames and the main window. When an object is dragged on top of a container that it can be placed into, the container will be highlighted to indicate that you can drop the widget inside. In addition, place twelve QPushButton widgets in the bottom frame.
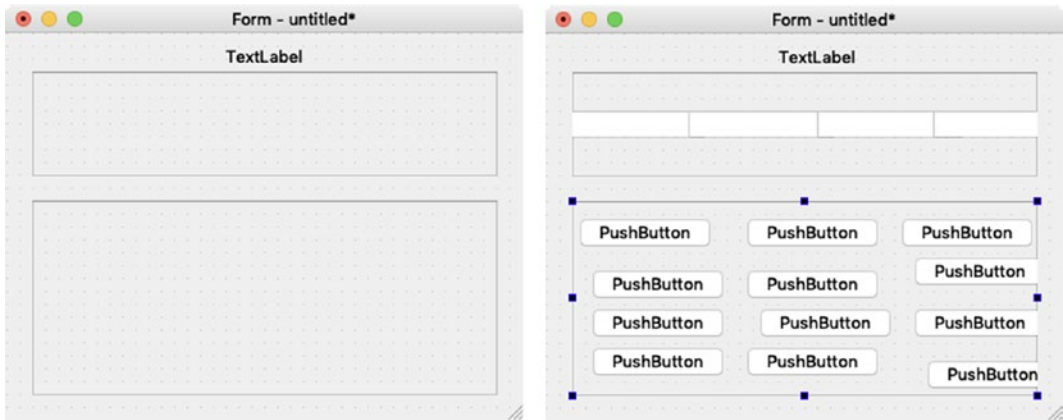
Figure 7-13. *The form with a label and two frames (left) and with the line edit widgets and push buttons added (right)*

The QFrame Class

The **QFrame** class is used as a container to group and surround widgets, or to act as placeholders in GUI applications. You can also apply a frame style to a QFrame container to visually separate it from nearby widgets. The following bit of code shows an example of how to create a frame object, modify its properties, and add a widget:

```
def frameUI(self):
    self.frame = QFrame(self) # Create QFrame object
    size_policy = QSizePolicy(QSizePolicy.Expanding, QSizePolicy.
    Preferred)

    self.frame.setSizePolicy(size_policy)
    self.frame.setFrameShape(QFrame.Box)
    self.frame.setFrameShadow(QFrame.Raised)
    self.frame.setLineWidth(3)
    self.frame.setMidLineWidth(5)

    # Set layout for QFrame object
    self.grid = QGridLayout(self.frame)

    # Place other widgets inside the frame by
    # calling the addWidget() method on the layout.
    self.button = QPushButton("Enter", self)
    self.grid.addWidget(self.button, 0, 0, 1, 0)
```

193

A frame object can have a number of different styles of frames, including boxes, panels, or lines. The style of the frame can be adjusted using the `setFrameShadow()`, `setLineWidth()`, and `setMidLineWidth()` methods.

Apply Layouts in Qt Designer

The next step is to add layouts to all of the containers and to the main window, as well. This is important to make sure that items are placed and resized correctly. Layouts can be added either from the toolbar or from context menus. It is possible to add more widgets to existing layouts once they have been set.

Since Qt Designer uses a drag and drop interface, you only need to place the objects on the form close to where you want them to be and then select one of the four layouts – QGridLayout, QHBoxLayout, QVBoxLayout, or QFormLayout – and Qt Designer will take care of placing them by using a widget's size hint. For more information about the types of layouts in PyQt, refer to Chapter 4.

Right-click the topmost frame to open a context menu. Scroll down to the last option, `Lay out`, and select `Layout Out Horizontally`. Do the same thing for the bottom frame, but this time select `Layout Out in a Grid`. This is demonstrated in Figure 7-14.
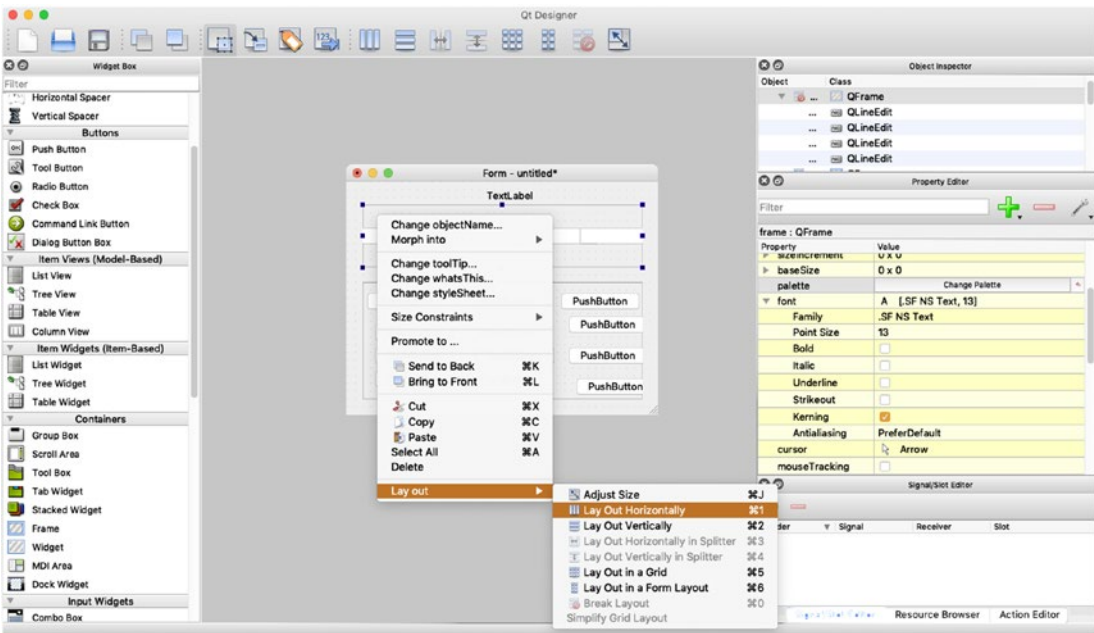

Figure 7-14. *Open a context menu to select layouts for containers and windows*

194

The top-level layout of a form can be set by right-clicking the form itself in the main window, and locating the layout you want to use. For the keypad GUI, select `Layout Out Vertically`. If the widgets are not aligned properly, you can also open the context menu, select `Break Layout`, and rearrange the widgets. Figure 7-15 shows the form with layouts applied.
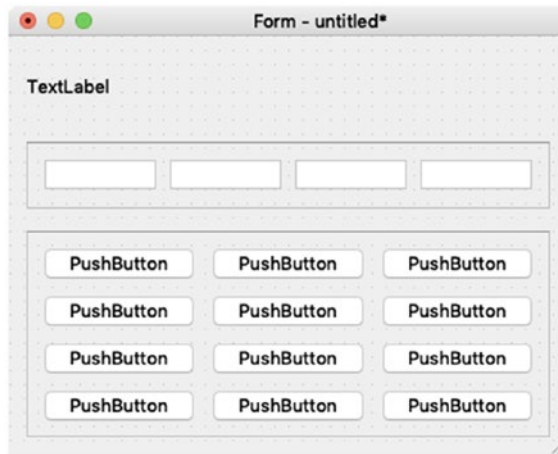

Figure 7-15. *The keypad GUI with layouts*

Edit the Properties of Objects

Once you have the layouts prepared, you should begin editing the features of the objects. This step could also be accomplished earlier when you place objects on the form.

The `Property Editor` is shown in Figure 7-4. It is organized into two columns, `Property` and `Value`. The properties are organized by Qt Classes.

To access and make changes to specific containers, widgets, layouts, or even the main window, you can click them in the form or in the `Object Inspector`. If a property is edited in the `Property Editor`, you can locate it by the following pattern:

`Qt Class` (Property column) ➤ `Property name` ➤ (submenu, if any) ➤ `Value`

The following are the steps that you can follow along to create the keypad GUI in Qt Designer:

1. Change window title: `QWidget` ➤ `windowTitle` ➤ `7.1` ➤ `Keypad GUI`

2. Double-click the QLabel. Change text to `Enter a passcode`.

3. Change QLabel properties:

   a. QWidget ➤ font ➤ font ➤ Point Size ➤ 20

   b. To edit palette colors, you will need to locate the palette property that opens a dialog box. Here you can change the colors for different parts of an object. To change the color of the text in the label object: QWidget ➤ palette ➤ Change Palette ➤ Window Text ➤ white

   c. QLabel ➤ alignment ➤ Horizontal ➤ AlignHCenter

4. Change top frame properties:

   a. QWidget ➤ sizePolicy ➤ Vertical Stretch ➤ 1

   b. QFrame ➤ frameShape ➤ NoFrame

   c. QFrame ➤ frameShadow ➤ Plain

5. For each of the four QLineEdit widgets, modify their properties:

   a. QWidget ➤ sizePolicy ➤ Vertical Policy ➤ Expanding

   b. QWidget ➤ font ➤ font ➤ Point Size ➤ 30

   c. QLineEdit ➤ alignment ➤ Horizontal ➤ AlignHCenter

6. Change bottom frame properties:

   a. QWidget ➤ sizePolicy ➤ Vertical Stretch ➤ 2

   b. QFrame ➤ frameShape ➤ Box

   c. QFrame ➤ frameShadow ➤ Sunken

   d. QFrame ➤ lineWidth ➤ 2

7. Double-click each of the buttons and change their text to 0–9, ∗, and #.

8. Edit each of the buttons' properties:

   a. QWidget ➤ sizePolicy ➤ Vertical Policy ➤ Expanding

   b. QWidget ➤ font ➤ font ➤ Point Size ➤ 36

9. Resize the main window:

   a. QWidget ➤ geometry ➤ Width ➤ 302

   b. QWidget ➤ geometry ➤ Height ➤ 406

10. Click the form and change its background color: `QWidget` ➤
    `palette` ➤ `Change Palette` ➤ `Window` ➤ `dark gray`

11. In the `Object Inspector`, double-click each of the default object
    names and edit them. The object name is used to reference the
    objects in the code.

After you have followed along with each of the steps, the form should look similar to
Figure 7-11.

Connect Signals and Slots in Qt Designer

Switch to the `Edit Signals/Slots` mode by selecting it from the toolbar. Qt Designer
has a simple interface for connecting signals and slots. Click the object that will emit a
signal and drag it to the object that will receive the signal, which is the slot.

For the keypad GUI, we are only making one set of connections. The remaining
signals and slots will be handled by manually coding them. When the '∗' button is
clicked, we want to clear all four line edit widgets. Click the button and drag the red
arrow to the first line edit object. A dialog box will appear (displayed in Figure 7-16) that
allows you to select the methods for both the signal and the slot. Select `clicked()` for the
button and `clear()` for the line edit. Finish connecting the other three line edit widgets.
Refer to Figure 7-17 as a guide for connecting the widgets.

Tip   When connecting signals and slots, make sure to check the "Show signals
and slots inherited from QWidget" checkbox to access more methods.

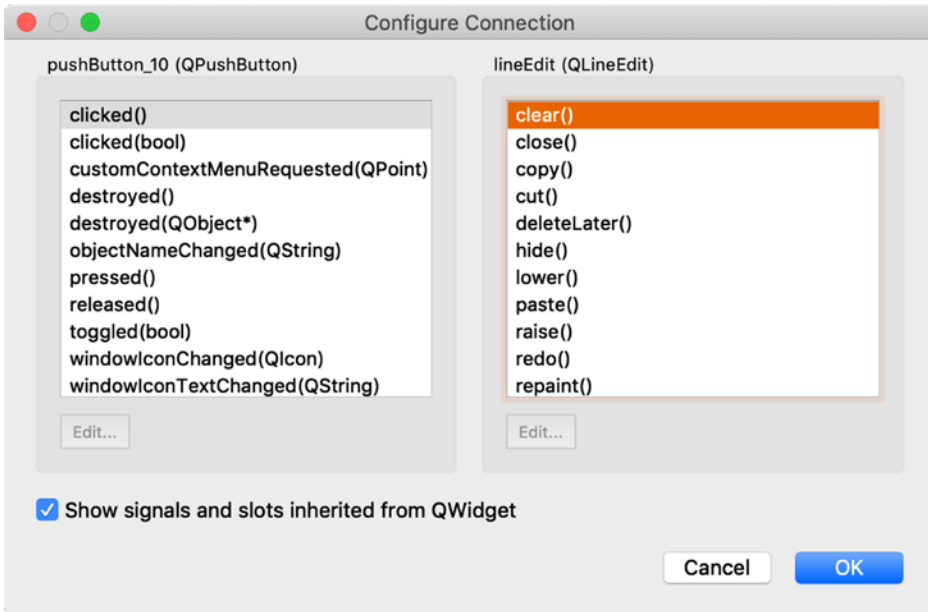
Figure 7-16. *The dialog box for connecting signals and slots*
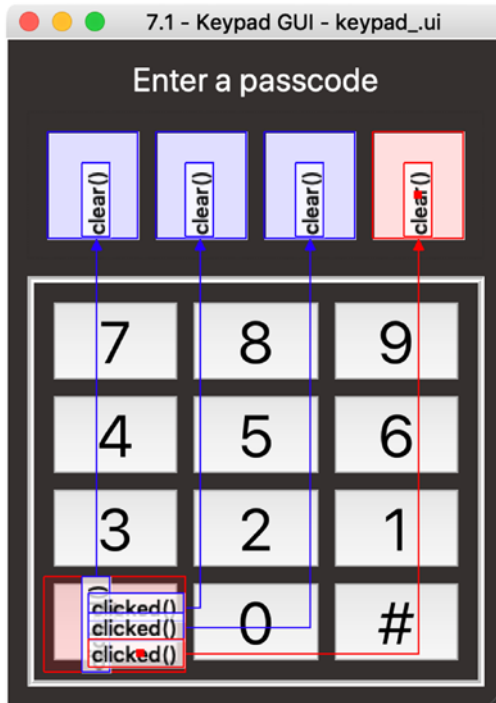

Figure 7-17. *The keypad GUI with signal and slot connections*

Preview Your GUI

It is often useful to view and interact with the form before exporting it to code. Not only can this be useful for checking the visual appearance of your GUI, but previewing also helps to make sure the signals and slots, resizing the window, and other functions are working properly.

To preview a form, open the Form menu and select Preview or use the hot keys Ctrl+R for Windows or Cmd+R for MacOS. If you are satisfied with your form, save it as a UI file with the .ui extension. Qt Designer UI files are written in XML format and contain the widget tree representation for creating a GUI.

Create and Edit Python Code

Qt Designer uses the Qt utility User Interface Compiler (uic) to generate code and create the user interface. However, since you are using PyQt5, you must use the uic Python module, pyuic5, to load .ui files and convert the XML file to Python code.

The pyuic5 utility is a command-line interface for interacting with uic. Open up the command prompt in Windows or Terminal in MacOS and navigate to the directory that contains the UI file. The format for converting to Python code is

```
pyuic5 filename.ui -o filename.py
```

To output a Python file, you need to include the -o option and the Python file to be written to, filename.py. This command will generate a single Python class. Generally, you will need to create a separate file to inherit from your newly created user interface class. Another option is to create an executable file that can display the GUI. This can be done by including the -x option.

```
pyuic5 -x filename.ui -o filename.py
```

Note   If you make changes to the GUI in Qt Designer after creating the Python file, you will need to call pyuic5 again on the UI file.

Tip   While it is possible to write new code in the newly generated Python file, the best thing to do would be to create a new file that imports the new user interface class. If you need to make changes to the GUI in Qt Designer and resave the file, it will erase any new code that you have written.

Extra Tips for Using Qt Designer

The following section briefly covers two additional topics:

- Creating GUIs with menus
- Displaying images in Qt Designer

Setting Up Main Windows and Menus

Open Qt Designer and select the Main Window template from the Form Menu in Figure 7-2. This creates a main window with a menubar and status bar by default. You can see a main window form displayed in Figure 7-1.

Adding Menus and Submenus in Qt Designer

Adding menus in Qt Designer is simple. Double-click the Type Here placeholder text in the menubar and enter the title of the menu. This process is shown in Figure 7-18. If you want to create a shortcut, you can also add the ampersand, '&', to the beginning of the menu's text. This updates the menubar object in the Object Inspector dialog. You can also edit the menu's properties in the Property Editor.
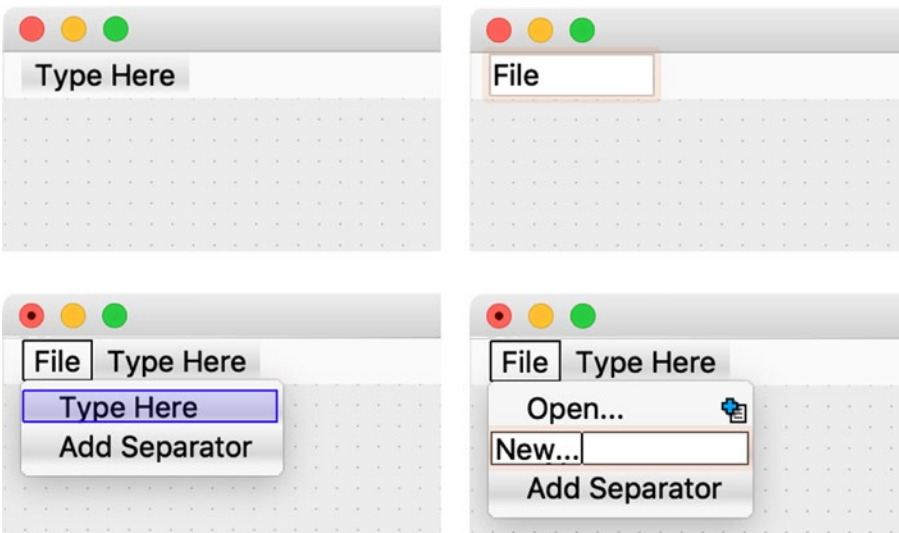

Figure 7-18. *Creating menus and menu entries. Type Here placeholder (top-left). Double-click the placeholder and enter menu's title (top-right). Add new menu entry (bottom-left). New menu entry (bottom-right)*

From here you can either add more menus, submenus, or actions. To add a submenu, first create a menu item. Then click the plus symbol next to the new entry in the menu. This will add a new menu that branches off of the existing menu entry. Double-click the Type Here placeholder and enter the text for the new item. Refer to Figure 7-19.
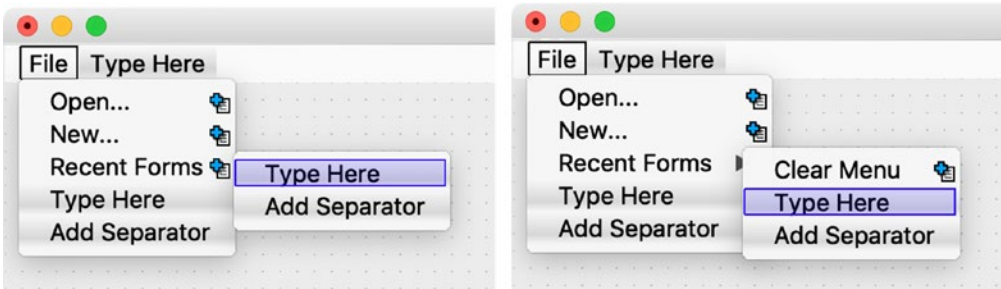

Figure 7-19. *Adding submenus. Click the plus symbol next to menu entry (left). Add new entry (right)*

Adding Toolbars in Qt Designer

Toolbars can be added to the main window by right-clicking the form to open a context menu. Click the Add Tool Bar option.

The actions in toolbars are created as toolbar buttons and can be dragged between the menus and the toolbar. You can also add icons to the toolbar. This topic is covered in the "Display Images in Qt Designer" section. An example of the toolbar with an icon is shown in Figure 7-20.
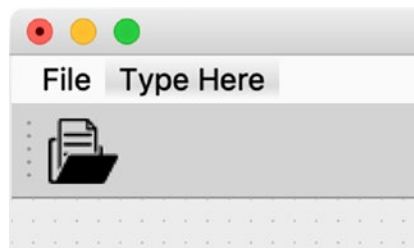

Figure 7-20. *Toolbar with Open toolbar button*

Adding Actions in Qt Designer

When items are first created in the menu and the toolbar, they are actually actions. Actions can be created, removed, given an icon, designated a shortcut hot key, and made checkable all in the Action Editor dock widget (shown in Figure 7-7). Actions can also be shared between the menu and the toolbar.

To share an action between the menu and the toolbar so that both objects contain the same item, drag the action from the Action Editor that already exists in the menu onto the toolbar.

Display Images in Qt Designer

This last section will take a quick look at how you can include images and icons in your application. Whether you are looking to add an image to a QLabel widget or trying to add icons to your toolbar, the process for adding an image is similar.

For example, if you have a QLabel widget on your form, you can access its properties in the Property Editor, shown below in Figure 7-21. Scroll down until you find the pixmap property. Click its Value and from here you will be able to search for an image file. If you want to add an icon, then you will look for the icon property, not pixmap.

You are given two options: Choose Resource... and Choose File.... If you have added resources to your project, then select Choose Resource.... Otherwise, you can search for images on your computer.
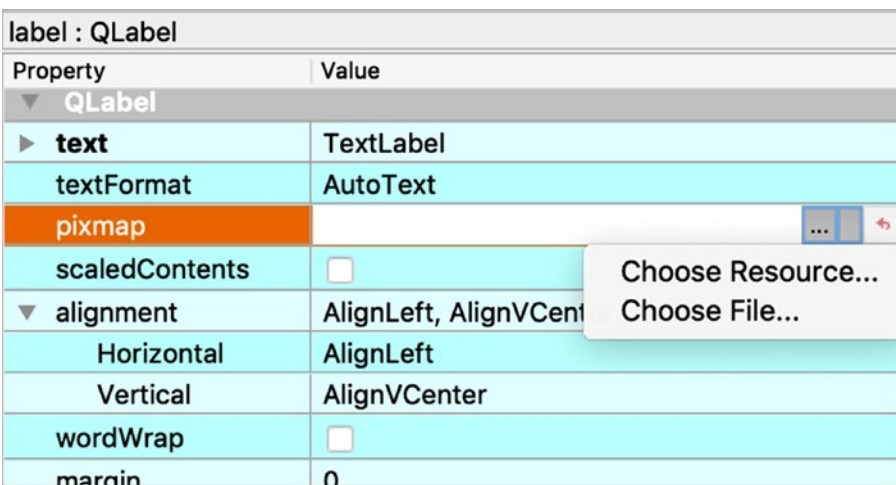

Figure 7-21. *Add images to your application using the pixmap property*

Summary

Qt Designer is definitely a useful tool for creating GUI applications. It provides a drag and drop interface that makes it easier to lay out widgets; modify the parameters of objects; create menus, toolbars, and dock widgets; add actions to menus; generate code that can be used in Python; and more. Qt Designer can make the design process much quicker and easier.

While this chapter covered a few of the basics for using Qt Designer, there are still other uses such as creating your own custom widgets that can be included in Qt Designer or generating dialog boxes.

The following chapters will begin to look at more specific classes that can be used to further augment a user interface. Chapter 8 takes a look at the QClipboard class and creating widgets with drag and drop functionality.

CHAPTER 8

Working with the Clipboard

One of the major benefits of GUIs is the ability to create programs that can visually interact with your system and other applications. Previous chapters have shown how to use dialog boxes to open and save files to your computer, change fonts or colors, and send images to a printer. In Chapter 8 you will see how to use the clipboard and drag and drop functions to extend the capabilities of your programs even further.

The **clipboard** is a location in your computer's memory that is used to temporarily store data that you have copied or cut from an application. The clipboard can store a number of different types of data, including text, images, gifs, and more. Information that is stored on your system's clipboard can be pasted into other applications, as long as it knows how to work with the type of data stored in the clipboard.

QClipboard is a PyQt class that is used to interact with your system's clipboard, allowing you to copy and paste between GUIs.

Chapter 8 introduces you to

- The `QClipboard` and `QMimeData` classes
- The Drag and Drop system in PyQt
- The `QListWidget` class for displaying item-based lists

The QClipboard Class

The **QClipboard** class makes your system's clipboard available so that you can copy and paste data such as text, images, and HTML text between applications. Qt widgets that can be used to manipulate textual information, such as QLineEdit, QTextEdit, and QListWidget, support using the clipboard. If you want to paste an image from the clipboard into an application, be sure to use widgets that support graphics, such as QLabel.

Including the clipboard in your project is pretty straightforward in PyQt. In order to access the QClipboard object in an application, create an instance of the clipboard by

```
clipboard = QApplication.clipboard()
```

To see one way to retrieve an image that has been copied to the clipboard, take a look at the following code:

```
self.label = QLabel() # Create label to hold image
self.clipboard = QApplication.clipboard() # Create cb object
self.label.setPixmap(self.clipboard.pixmap())
```

The process for text or HTML text is similar. Another way to get data is to use the QMimeData class which will be covered in the "Explanation" section of this example.

The events that occur between your system and an application built using PyQt are handled by QApplication. The clipboard gives you the ability to send or receive data in your application. This means that you can not only get data from other programs but can also send it out, as well. However, the clipboard can only hold one object at a time. So if you copy an image to the clipboard and then copy text, only the text will be available and the image will have been deleted.

In Listing 8-1 you will see how to set up the clipboard and actually be able to visualize its contents after copying text from another window.

Listing 8-1. Example code that demonstrates how to use the clipboard

```
# clipboard_ex.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QPushButton,
QTextEdit, QDockWidget, QVBoxLayout, QFrame)
from PyQt5.QtCore import Qt, QSize

class ClipboardEx(QMainWindow):
    def __init__(self):
        super().__init__()

        self.initializeUI()
```

```python
    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setMinimumSize(QSize(500, 300))
        self.setWindowTitle("Clipboard Example")

        self.central_widget = QTextEdit()
        self.setCentralWidget(self.central_widget)

        self.createClipboard()

        self.show()

    def createClipboard(self):
        """
        Set up clipboard and dock widget to display text from
        the clipboard.
        """
        # Create dock widget
        clipboard_dock = QDockWidget()
        clipboard_dock.setWindowTitle("Display Clipboard Contents")
        clipboard_dock.setAllowedAreas(Qt.TopDockWidgetArea)

        dock_frame = QFrame()
        self.cb_text = QTextEdit()
        paste_button = QPushButton("Paste")
        paste_button.clicked.connect(self.pasteText)

        dock_v_box = QVBoxLayout()
        dock_v_box.addWidget(self.cb_text)
        dock_v_box.addWidget(paste_button)

        # Set the main layout for the dock widget,
        # then set the main widget of the dock widget
        dock_frame.setLayout(dock_v_box)
        clipboard_dock.setWidget(dock_frame)
```

```python
        # Set initial location of dock widget
        self.addDockWidget(Qt.TopDockWidgetArea, clipboard_dock)

        # Create instance of the clipboard
        self.clipboard = QApplication.clipboard()
        self.clipboard.dataChanged.connect(self.copyFromClipboard)

    def copyFromClipboard(self):
        """
        Get the contents of the system clipboard.
        """
        mime_data = self.clipboard.mimeData()
        if mime_data.hasText():
            self.cb_text.setText(mime_data.text())
            #self.cb_text.repaint() # Uncomment if text not updating

    def pasteText(self):
        """
        Paste text from the clipboard if button is clicked.
        """
        self.central_widget.paste()
        #self.central_widget.repaint() # Uncomment if text not updating

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = ClipboardEx()
    sys.exit(app.exec_())
```

The finished application can be seen in Figure 8-1. The top text edit widget displays the current contents of the clipboard. The user can then paste it into the main window which is the lower text edit widget.
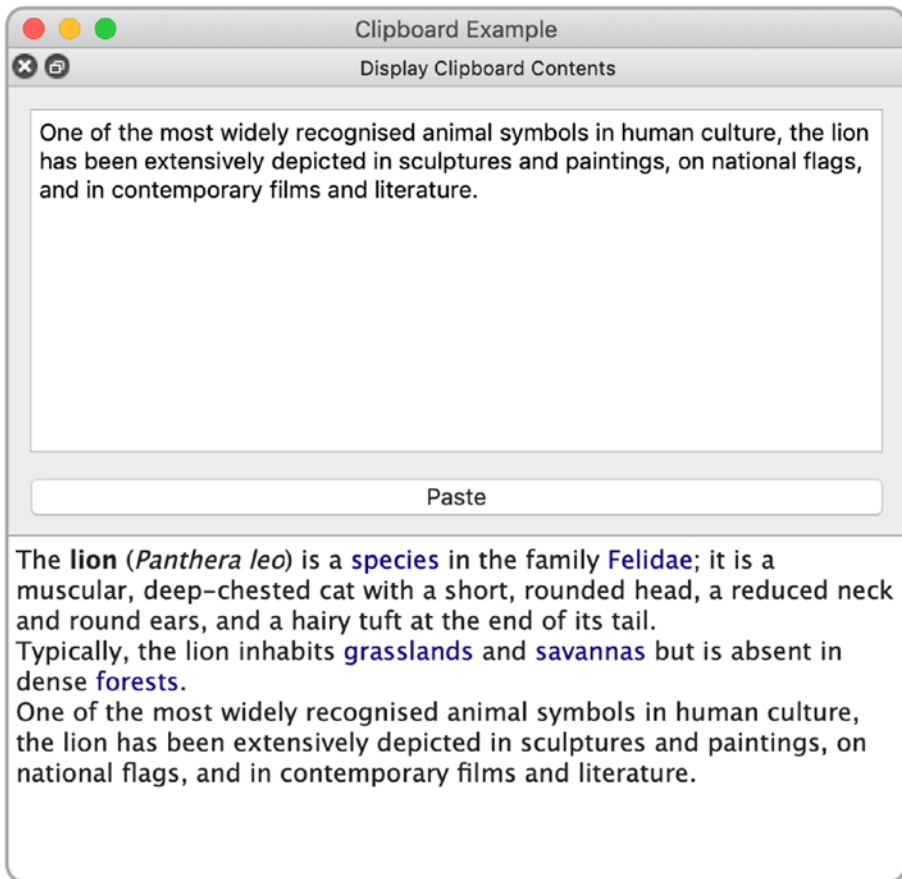
Figure 8-1. *Example GUI that allows the user to see the contents of the clipboard in the top dock widget*

Explanation

In some applications you may actually want to see the contents of the clipboard in a separate window before pasting it into the main window. A dock widget, especially one that can float separate from the main window, is perfect to use as a clipboard manager.

After importing classes and setting up the window, we set the central_widget of the main window to be a QTextEdit widget. The central widget is where we will edit the text that is pasted from the clipboard.

The `clipboard_dock` widget is set up so that it can either float or be attached to the top of the main window. We need to use a `QFrame` container to hold the `cb_text` and `paste_button` widgets. If new text is copied from another application, then `cb_text` will display the text. If the user wants to retain the text, then they can paste it into the `central_widget`.

Using a single line, PyQt makes it simple to include the clipboard in an application.

```
self.clipboard = QApplication.clipboard() # Create the clipboard object
self.clipboard.dataChanged.connect(self.copyFromClipboard)
```

The `dataChanged()` method emits a signal if the contents of the clipboard have changed. If a change has occurred, then the `cb_text` widget is updated to display the new clipboard text using the `copyFromClipboard()` method. To check what kind of data is stored in the clipboard, we use the QMimeData class.

The **QMimeData** class is used for both the clipboard and the drag and drop system in PyQt, the latter of which will be introduced later in this chapter. The Multipurpose Internet Mail Extensions (MIME) format supports not only text but HTML, URLs, images, and color data, as well. Objects created from the QMimeData class ensure that information can be safely moved between applications and also between objects in the same application.

```
mime_data = self.clipboard.mimeData()
```

The method `mimeData()` returns information about the data currently in the clipboard. To check if the object can return plain text, we use the `hasText()` method. If the data is text, then we get the text using `mime_data.text()` and set the text of the QTextEdit widget using `setText()`. A similar process is also used to access other kinds of data using QMimeData.

Finally, the QTextEdit method `paste()` is called in `pasteText()` to fetch the text in the clipboard if the button is pressed. The `repaint()` method is used to force the text of the widget to update.

Project 8.1 – Sticky Notes GUI

Sometimes you have an idea, a note, or a bit of information that you need to quickly jot down. Maybe you need to remind yourself of an appointment and need to make a note to yourself. You only need a small, temporary, maybe even colorful, area to help brainstorm and organize those ideas. Sticky notes are perfect for those uses and more.

The sticky notes GUI, shown in Figure 8-2, allows you to open as many windows as you want. You can edit the text of each note individually, change the color of a note, and also paste text from the clipboard. This project demonstrates a practical use for the clipboard class and acts as a foundation if you choose to build your own sticky notes application.
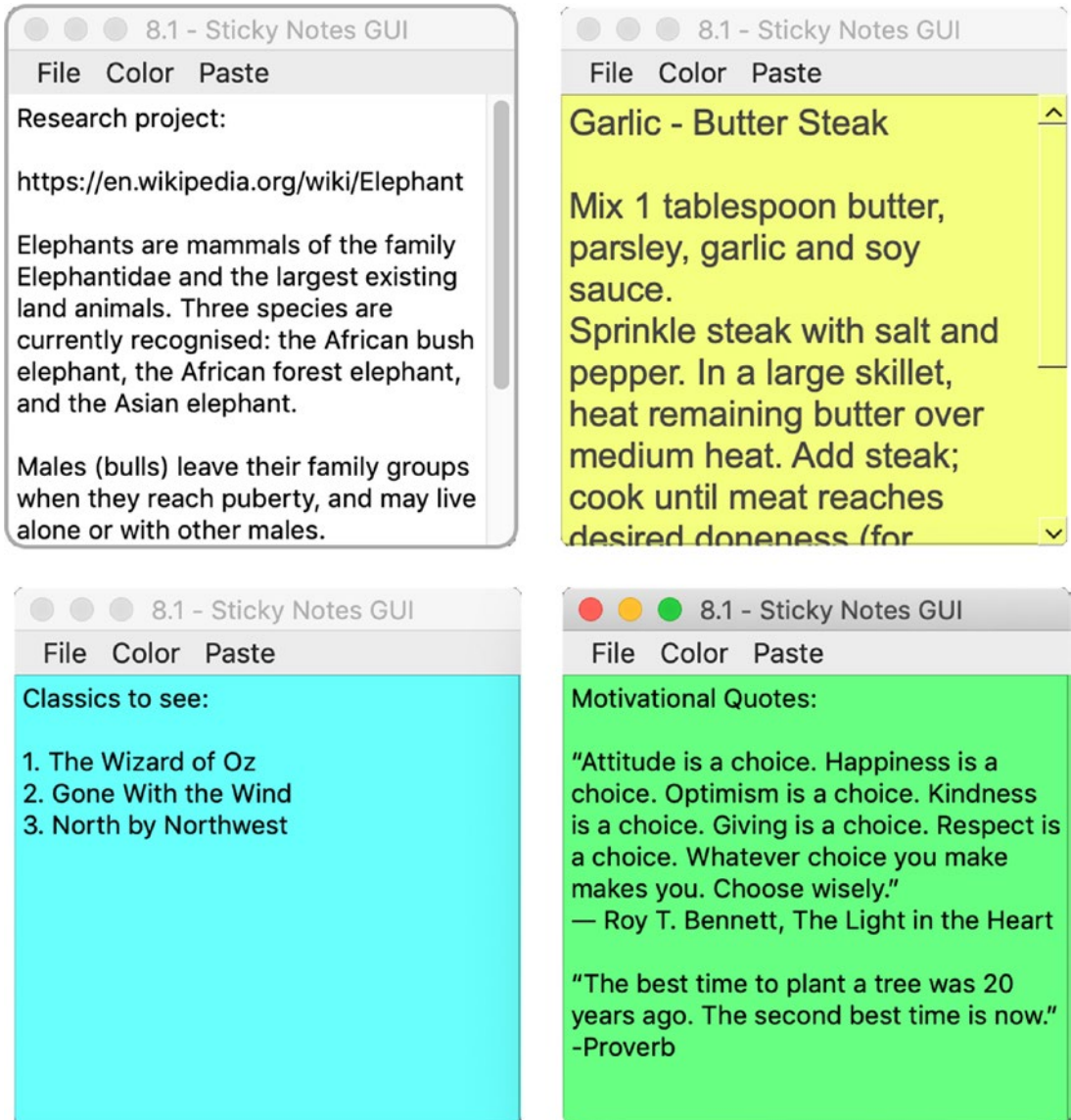

Figure 8-2. *The sticky notes GUI*

Sticky Notes GUI Solution

The sticky notes GUI is relatively simple, created of a single QTextEdit widget that serves as the central widget of the main window (Listing 8-2). The menu system allows you to create a new note, clear the text in the text edit widget, quit the application, change the background color, and paste text from the clipboard.

Listing 8-2.   Code for sticky notes GUI

```
# stickynotes.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QAction, QTextEdit)
from PyQt5.QtCore import QSize

class StickyNotes(QMainWindow):
    # Static variables
    note_id = 1
    notes = []

    def __init__(self, note_ref=str()):
        super().__init__()
        self.note_ref = note_ref
        StickyNotes.notes.append(self)

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setMinimumSize(QSize(250, 250))
        self.setWindowTitle("8.1 - Sticky Notes GUI")

        self.central_widget = QTextEdit()
        self.setCentralWidget(self.central_widget)
```

```python
        self.createMenu()
        self.createClipboard()

        self.show()

    def createMenu(self):
        """
        Create simple menu bar and menu actions
        """
        # Create actions for the file menu
        self.new_note_act = QAction('New Note', self)
        self.new_note_act.setShortcut('Ctrl+N')
        self.new_note_act.triggered.connect(self.newNote)

        self.close_act = QAction('Clear', self)
        self.close_act.setShortcut('Ctrl+W')
        self.close_act.triggered.connect(self.clearNote)

        self.quit_act = QAction('Quit', self)
        self.quit_act.setShortcut('Ctrl+Q')
        self.quit_act.triggered.connect(self.close)

        # Create actions for the color menu
        self.yellow_act = QAction('Yellow', self)
        self.yellow_act.triggered.connect(lambda: self.
        changeBackground(self.yellow_act.text()))

        self.blue_act = QAction('Blue', self)
        self.blue_act.triggered.connect(lambda: self.changeBackground(self.
        blue_act.text()))

        self.green_act = QAction('Green', self)
        self.green_act.triggered.connect(lambda: self.
        changeBackground(self.green_act.text()))

        # Create actions for the paste menu
        self.paste_act = QAction('Paste', self)
        self.paste_act.setShortcut('Ctrl+V')
        self.paste_act.triggered.connect(self.pasteToClipboard)
```

213

```python
        # Create menubar
        menu_bar = self.menuBar()
        menu_bar.setNativeMenuBar(False) # Uncomment to display menu in
        window on MacOS

        # Create file menu and add actions
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(self.new_note_act)
        file_menu.addAction(self.close_act)
        file_menu.addAction(self.quit_act)

        # Create color menu and add actions
        file_menu = menu_bar.addMenu('Color')
        file_menu.addAction(self.yellow_act)
        file_menu.addAction(self.blue_act)
        file_menu.addAction(self.green_act)

        # Create paste menu and add actions
        file_menu = menu_bar.addMenu('Paste')
        file_menu.addAction(self.paste_act)

    def createClipboard(self):
        """
        Set up clipboard.
        """
        self.clipboard = QApplication.clipboard()
        self.clipboard.dataChanged.connect(self.copyToClipboard)

    def newNote(self):
        """
        Create new instance of StickyNotes class.
        """
        self.note_ref = str("note_%d" % StickyNotes.note_id)
        StickyNotes().show()
        StickyNotes.note_id += 1

    def clearNote(self):
        """
```

```python
        Delete the current note's text.
        """
        self.central_widget.clear()

    def copyToClipboard(self):
        """
        Get the contents of the system clipboard.
        """
        return self.clipboard.text()

    def pasteToClipboard(self):
        """
        Get the contents of the system clipboard and paste
        into the note.
        """
        text = self.copyToClipboard()
        self.central_widget.insertPlainText(text + '\n')

    def changeBackground(self, color_text):
        """
        Change a note's background color.
        """
        if color_text == "Yellow":
            self.central_widget.setStyleSheet("background-color: rgb(248,
            253, 145)")
        elif color_text == "Blue":
            self.central_widget.setStyleSheet("background-color: rgb(145,
            253, 251)")
        elif color_text == "Green":
            self.central_widget.setStyleSheet("background-color: rgb(148,
            253, 145)")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = StickyNotes()
    sys.exit(app.exec_())
```

The completed project can be seen in Figure 8-2.

Explanation

The sticky notes GUI is a good project to introduce the concept of **single-document interface** (**SDI**). SDI is a method that organizes GUIs into individual windows that are handled separately. Even though the sticky notes application allows you to create multiple instances of the GUI at the same time, each window is separate and independent from the others. The contrast is **multiple-document interface** (**MDI**), where a single parent window contains and controls multiple nested child windows. An example of MDI can be found in Chapter 12.

We create the StickyNotes class for the GUI and first include two static variables, note_id, used to give a unique name to each new window, and notes, to keep track of the many different windows by appending them to a list. The static variables are shared by all instances of the class.

The menu system is set up just like in previous chapters. The Color menu allows the user to select a background color for each note. If the user wants to paste text from the clipboard into a widget, they can either use the Paste menu entry or the hot key Ctrl+V.

Let's take a look at the different class methods. The clipboard object is created and updated using the dataChanged() signal in createClipboard(). Each new note is given a new name, note_ref, when they are created based on the current note_id value. The other functions allow you to interact with the text and clipboard or edit the background color of the central_widget.

Drag and Drop in PyQt

The drag and drop mechanism allows a user to perform tasks in a GUI by selecting items, such as icons or images, and move them into another window or onto another object. PyQt also makes including this behavior in an application very simple, as well. To allow widgets to have drag and drop functionality, you only need to set their setAcceptDrops() and setDragEnabled() properties to True.

With drag and drop functionality enabled, you can move items from one text edit, list, or table object to another in PyQt. QMimeData can also be used to handle custom data types.

Listing 8-3 illustrates how to drag and drop icons between two QListWidget objects in the same GUI window.

Listing 8-3. Code that demonstrates an example of drag and drop

```python
# drag_drop.py
# Import necessary modules
import sys, os
from PyQt5.QtWidgets import (QApplication, QWidget, QListWidget, QLabel,
QGridLayout, QListWidgetItem)
from PyQt5.QtCore import QSize
from PyQt5.QtGui import QIcon

class DragAndDropGUI(QWidget):
    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 500, 300)
        self.setWindowTitle("Drag and Drop Example")
        self.setupWidgets()

        self.show()

    def setupWidgets(self):
        """
        Create and arrange widgets in window
        """
        icon_label = QLabel("ICONS", self)
        icon_widget = QListWidget()
        icon_widget.setAcceptDrops(True)
        icon_widget.setDragEnabled(True)
        icon_widget.setViewMode(QListWidget.IconMode)
```

```python
        image_path = "images"
        for img in os.listdir(image_path):
            list_item = QListWidgetItem()
            list_item.setText(img.split(".")[0])
            list_item.setIcon(QIcon(os.path.join(image_path,
                              "{0}").format(img)))
            icon_widget.setIconSize(QSize(50, 50))
            icon_widget.addItem(list_item)

        list_label = QLabel("LIST", self)
        list_widget = QListWidget()
        list_widget.setAlternatingRowColors(True)
        list_widget.setAcceptDrops(True)
        list_widget.setDragEnabled(True)

        # Create grid layout
        grid = QGridLayout()
        grid.addWidget(icon_label, 0, 0)
        grid.addWidget(list_label, 0, 1)
        grid.addWidget(icon_widget, 1, 0)
        grid.addWidget(list_widget, 1, 1)

        self.setLayout(grid)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = DragAndDropGUI()
    sys.exit(app.exec_())
```

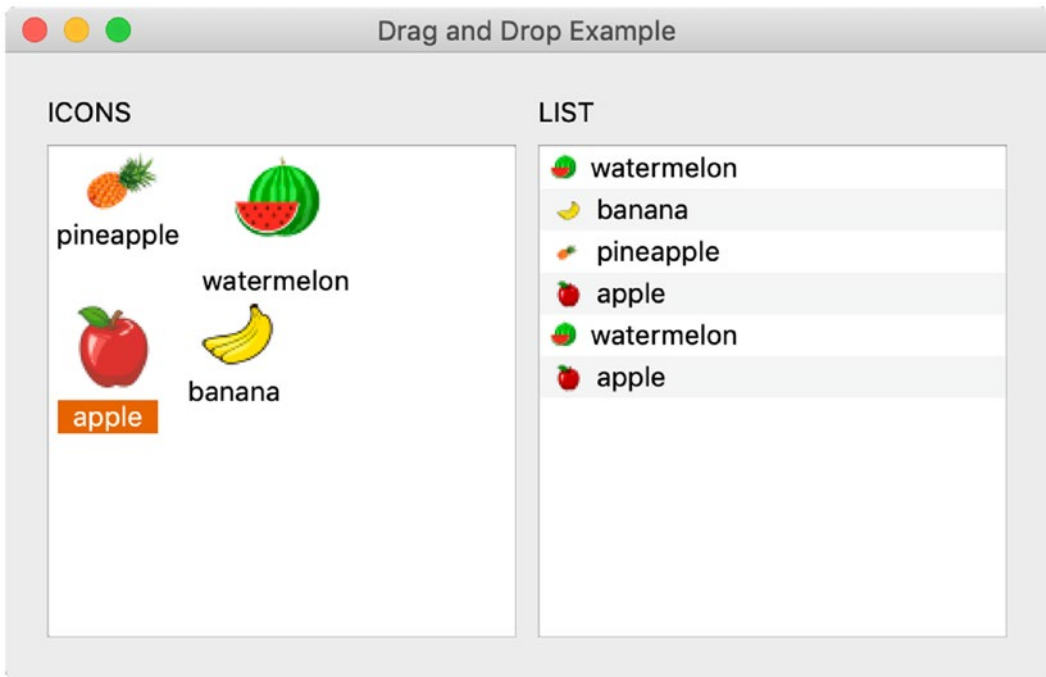The drag and drop GUI can be seen in Figure 8-3.

Figure 8-3. *Two QListWidget objects used to demonstrate drag and drop*

Explanation

There are two instances of the QListWidget class created, icon_widget and list_widget. The icon_widget object displays objects in IconMode, set by

```
icon_widget.setViewMode(QListWidget.IconMode)
```

The QListWidget default setting shows items in a list. To set up the drag and drop capability for icon_widget, call the setAcceptDrops() and setDragEnabled() methods. Then repeat the process for list_widget.

```
icon_widget = QListWidget()
icon_widget.setAcceptDrops(True)
icon_widget.setDragEnabled(True)
```

When one of the icons that are loaded into the icon_widget is dragged onto the list_widget, the list updates its contents to include the new item. Dropping an item from one QListWidget to the other adds a new item to that list.

The next section introduces the QListWidget class and a few of its methods.

The QListWidget Class

The **QListWidget** class creates a widget with an item-based interface that makes it simpler for adding and removing items. Items can be added either when the widget is created in code or added on later. The **QListWidgetItem** class is used in conjunction with QListWidget to serve as an item that can be used with the list. In the previous example, `list_item` creates an item that includes text and an icon to be added to the list using the `addItem()` method.

Listing 8-4 briefly demonstrates how to add, insert, remove, and clear all items from a QListWidget.

Listing 8-4. An example of using the QListWidget class to manage items in a list

```python
# listwidget_ex.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QListWidget,
QPushButton, QHBoxLayout, QVBoxLayout, QListWidgetItem, QInputDialog)

class GroceryListGUI(QWidget):
    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 400, 200)
        self.setWindowTitle("QListWidget Example")
        self.setupWidgets()

        self.show()

    def setupWidgets(self):
        """
        Create and arrange widgets in window
        """
```

```python
self.list_widget = QListWidget()
self.list_widget.setAlternatingRowColors(True)

# Initialize the QlistWidget with items
grocery_list = ["grapes", "broccoli", "garlic", "cheese", "bacon",
"eggs", "waffles", "rice", "soda"]
for item in grocery_list:
    list_item = QListWidgetItem()
    list_item.setText(item)
    self.list_widget.addItem(list_item)

# Create buttons
add_button = QPushButton("Add")
add_button.clicked.connect(self.addListItem)

insert_button = QPushButton("Insert")
insert_button.clicked.connect(self.insertItemInList)

remove_button = QPushButton("Remove")
remove_button.clicked.connect(self.removeOneItem)

clear_button = QPushButton("Clear")
clear_button.clicked.connect(self.list_widget.clear)

# Create layout
right_v_box = QVBoxLayout()
right_v_box.addWidget(add_button)
right_v_box.addWidget(insert_button)
right_v_box.addWidget(remove_button)
right_v_box.addWidget(clear_button)

left_h_box = QHBoxLayout()
left_h_box.addWidget(self.list_widget)
left_h_box.addLayout(right_v_box)

self.setLayout(left_h_box)
```

```python
    def addListItem(self):
        """
        Add a single item to the list widget.
        """
        text, ok = QInputDialog.getText(self, "New Item", "Add item:")
        if ok and text != "":
            list_item = QListWidgetItem()
            list_item.setText(text)
            self.list_widget.addItem(list_item)

    def insertItemInList(self):
        """
        Insert a single item into the list widget under the
        current highlighted row.
        """
        text, ok = QInputDialog.getText(self, "Insert Item", "Insert item:")
        if ok and text != "":
            row = self.list_widget.currentRow()
            row = row + 1 # select row below current row
            new_item = QListWidgetItem()
            new_item.setText(text)
            self.list_widget.insertItem(row, new_item)

    def removeOneItem(self):
        """
        Remove a single item from the list widget.
        """
        row = self.list_widget.currentRow()
        self.list_widget.takeItem(row)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = GroceryListGUI()
    sys.exit(app.exec_())
```

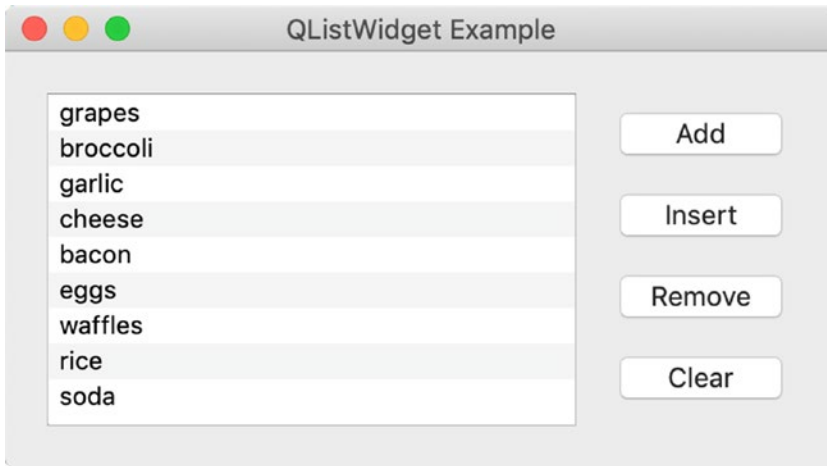Your GUI should look similar to the one in Figure 8-4.

Figure 8-4. *QListWidget could be used to display objects in an inventory or items in a directory*

Explanation

QListWidget allows you to manage the items displayed in the GUI window. For alternating row colors, set the `setAlternatingRowColors()` methods to `True`.

The `list_widget` object is populated with items from the list when the program begins. Using the buttons on the right, the user can edit the list items. Each method in the `GroceryListGUI` class is used to demonstrate one of the following `QListWidget` methods:

- `addItem()` – Add an item to the end of a list

- `insertItem()` – Insert an item at the specified row

- `takeItem()` – Remove an item from the specified row

- `clear()` – Remove all items from the list

Summary

The QClipboard class allows GUI applications to receive and send data from the system's clipboard. Drag and drop is another type of functionality that GUIs can utilize to pass data between widgets and other programs. Drag and drop is very simple to include in your own projects with only a few lines of code. The QMimeData class handles various kinds of data types for both clipboard and drag and drop systems, ensuring proper data handling.

Many of PyQt's widgets for editing text already include the ability to interact with the clipboard, so you won't often need to include the code for the clipboard in your program.

In the next chapter, we will see how to implement animation and graphics into PyQt applications and learn how to create custom widgets.

Graphics and Animation in PyQt

After going through many examples in previous chapters that introduce you to the fundamentals for building GUIs, Chapter 9 finally allows for you to explore your creative and artistic side through drawing and animation in PyQt5.

Graphics in PyQt is done primarily with the **QPainter** API. PyQt's painting system handles drawing for text, images, and vector graphics and can be done on a variety of surfaces, including QImage, QWidget, and QPrinter. With QPainter you can enhance the look of existing widgets or create your own.

The main components of the painting system in PyQt are the **QPainter**, **QPaintDevice**, and **QPaintEngine** classes. QPainter performs the drawing operations; a QPaintDevice is an abstraction of two-dimensional space which acts as the surface that QPainter can paint on; and QPaintEngine is the internal interface used by the QPainter and QPaintDevice classes for drawing.

In this chapter, we are going to be taking a look at 2D graphics, covering topics such as drawing simple lines and shapes, designing your own painting application, and animation. If you are interested in creating GUIs that work with 3D visuals, Qt also has support for OpenGL, which is software that renders both 2D and 3D graphics.

New concepts introduced in this chapter include

- An introduction to `QPainter` and other classes used for drawing PyQt

- Creating tool tips using `QToolTip`

- Animating objects using `QPropertyAnimation` and `pyqtProperty`

- How to set up a Graphics View and a Graphics Scene for interacting with items in a GUI window

- A new widget for selecting values in a bounded range, `QSlider`

- • Handling mouse events with event handlers

- • PyQt's four image handling classes

- • Creating your own custom widgets using PyQt

Introduction to the QPainter Class

Whenever you need to draw something in PyQt, you will more than likely need to work with the QPainter class. QPainter provides functions for drawing simple points and lines, complex shapes, text, and pixmaps. We have looked at pixmaps in previous chapters in applications where we needed to display images. QPainter also allows you to customize a variety of its settings, such as rendering quality or changing the painter's coordinate system. Drawing can be done on a **paint device**, which are two-dimensional objects created from the different PyQt classes that can be painted on with QPainter.

Drawing relies on a coordinate system for specifying the position of points and shapes and is typically handled in the paint event of a widget. The default coordinate system for a paint device has the origin at the top-left corner, beginning at (0, 0). The x values increase to the right and y values increase going down. Each (x, y) coordinate defines the location of a single pixel.

The following example illustrates a few of the drawing functions and shows how to use the QPen and QBrush classes and how to set up the paintEvent() function for drawing on a widget.

Listing 9-1. This code gives examples for drawing with the QPainter class

```
# paint_basics.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtGui import (QPainter, QPainterPath, QColor, QBrush, QPen,
QFont, QPolygon, QLinearGradient)
from PyQt5.QtCore import Qt, QPoint, QRect
```

```python
class Drawing(QWidget):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setFixedSize(600, 600)
        self.setWindowTitle('QPainter Basics')

        # Create a few pen colors
        self.black = '#000000'
        self.blue = '#2041F1'
        self.green = '#12A708'
        self.purple = '#6512F0'
        self.red = '#E00C0C'
        self.orange = '#FF930A'

        self.show()

    def paintEvent(self, event):
        """
        Create QPainter object and handle paint events.
        """
        painter = QPainter()
        painter.begin(self)
        # Use antialiasing to smooth curved edges
        painter.setRenderHint(QPainter.Antialiasing)

        self.drawPoints(painter)
        self.drawDiffLines(painter)
        self.drawText(painter)
        self.drawRectangles(painter)
        self.drawPolygons(painter)
```

```python
        self.drawRoundedRects(painter)
        self.drawCurves(painter)
        self.drawCircles(painter)
        self.drawGradients(painter)

        painter.end()

    def drawPoints(self, painter):
        """
        Example of how to draw points with QPainter.
        """
        pen = QPen(QColor(self.black))

        for i in range(1, 9):
            pen.setWidth(i * 2)
            painter.setPen(pen)
            painter.drawPoint(i * 20, i * 20)

    def drawDiffLines(self, painter):
        """
        Examples of how to draw lines with QPainter.
        """
        pen = QPen(QColor(self.black), 2)

        painter.setPen(pen)
        painter.drawLine(230, 20, 230, 180)

        pen.setStyle(Qt.DashLine)
        painter.setPen(pen)
        painter.drawLine(260, 20, 260, 180)

        pen.setStyle(Qt.DotLine)
        painter.setPen(pen)
        painter.drawLine(290, 20, 290, 180)

        pen.setStyle(Qt.DashDotLine)
        painter.setPen(pen)
        painter.drawLine(320, 20, 320, 180)
```

```python
        # Change the color and thickness of the pen
        blue_pen = QPen(QColor(self.blue), 4)

        painter.setPen(blue_pen)
        painter.drawLine(350, 20, 350, 180)

        blue_pen.setStyle(Qt.DashDotDotLine)
        painter.setPen(blue_pen)
        painter.drawLine(380, 20, 380, 180)

    def drawText(self, painter):
        """
        Example of how to draw text with QPainter.
        """
        text = "Don't look behind you."

        pen = QPen(QColor(self.red))
        painter.setFont(QFont("Helvetica", 15))
        painter.setPen(pen)
        painter.drawText(420, 110, text)

    def drawRectangles(self, painter):
        """
        Examples of how to draw rectangles with QPainter.
        """
        pen = QPen(QColor(self.black))
        brush = QBrush(QColor(self.black))

        painter.setPen(pen)
        painter.drawRect(20, 220, 80, 80)

        painter.setPen(pen)
        painter.setBrush(brush)
        painter.drawRect(120, 220, 80, 80)

        red_pen = QPen(QColor(self.red), 5)
        green_brush = QBrush(QColor(self.green))
```

```python
        painter.setPen(red_pen)
        painter.setBrush(green_brush)
        painter.drawRect(20, 320, 80, 80)

        # Demonstrate how to change the alpha channel
        # to include transparency
        blue_pen = QPen(QColor(32, 85, 230, 100), 5)
        blue_pen.setStyle(Qt.DashLine)
        painter.setPen(blue_pen)
        painter.setBrush(green_brush)
        painter.drawRect(120, 320, 80, 80)

    def drawPolygons(self, painter):
        """
        Example of how to draw polygons with QPainter.
        """
        pen = QPen(QColor(self.blue), 2)
        brush = QBrush(QColor(self.orange))

        points = QPolygon([QPoint(240, 240), QPoint(380, 250),
                        QPoint(230, 380), QPoint(370, 360)])

        painter.setPen(pen)
        painter.setBrush(brush)
        painter.drawPolygon(points)

    def drawRoundedRects(self, painter):
        """
        Examples of how to draw rectangles with
        rounded corners with QPainter.
        """
        pen = QPen(QColor(self.black))
        brush = QBrush(QColor(self.black))

        rect_1 = QRect(420, 340, 40, 60)
        rect_2 = QRect(480, 300, 50, 40)
        rect_3 = QRect(540, 240, 40, 60)
```

```python
        painter.setPen(pen)
        brush.setStyle(Qt.Dense1Pattern)
        painter.setBrush(brush)
        painter.drawRoundedRect(rect_1, 8, 8)

        brush.setStyle(Qt.Dense5Pattern)
        painter.setBrush(brush)
        painter.drawRoundedRect(rect_2, 5, 20)

        brush.setStyle(Qt.BDiagPattern)
        painter.setBrush(brush)
        painter.drawRoundedRect(rect_3, 15, 15)

    def drawCurves(self, painter):
        """
        Examples of how to draw curves with QPainterPath.
        """
        pen = QPen(Qt.black, 3)
        brush = QBrush(Qt.white)

        path = QPainterPath()
        path.moveTo(30, 420)
        path.cubicTo(30, 420, 65, 500, 30, 560)
        path.lineTo(163, 540)
        path.cubicTo(125, 360, 110, 440, 30, 420)
        path.closeSubpath()

        painter.setPen(pen)
        painter.setBrush(brush)
        painter.drawPath(path)

    def drawCircles(self, painter):
        """
        Example of how to draw ellipses with QPainter.
        """
        height, width = self.height(), self.width()

        center_x, center_y = (width / 2), height - 100
        radius_x, radius_y = 60, 60
```

```python
        pen = QPen(Qt.black, 2, Qt.SolidLine)
        brush = QBrush(Qt.darkMagenta, Qt.Dense5Pattern)

        painter.setPen(pen)
        painter.setBrush(brush)
        painter.drawEllipse(QPoint(center_x, center_y), radius_x, radius_y)

    def drawGradients(self, painter):
        """
        Example of how to draw fill shapes using gradients.
        """
        pen = QPen(QColor(self.black), 2)
        gradient = QLinearGradient(450, 480, 520, 550)

        gradient.setColorAt(0.0, Qt.blue)
        gradient.setColorAt(0.5, Qt.yellow)
        gradient.setColorAt(1.0, Qt.cyan)

        painter.setPen(pen)
        painter.setBrush(QBrush(gradient))
        painter.drawRect(420, 420, 160, 160)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = Drawing()
    sys.exit(app.exec_())
```

The results of different QPainter drawing functions can be seen in Figure 9-1.
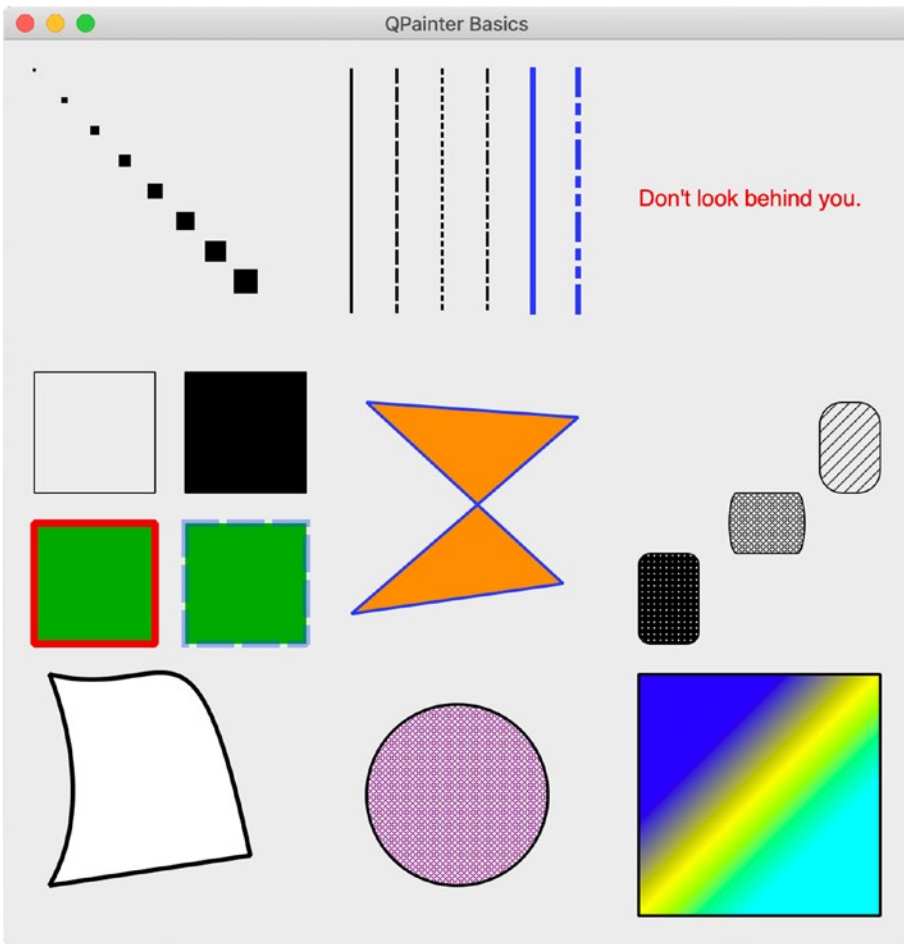
Figure 9-1. *The window illustrates a few of the QPainter class's different functions. Starting from the top-left corner, the first row presents points, lines, and text; the second row illustrates shapes and patterns, including rectangles, polygons, and rectangles with rounded corners; the last row displays drawing curves, circles, and painting with gradients*

Explanation

This program introduces quite a few new classes, a majority of them imported from the QtGui module. QtGui provides us with the tools we need for 2D graphics, imaging, and fonts. The QPoint and QRect classes imported from QtCore are used to define points and rectangles specified by coordinate values in the window's plane.

The Drawing class inherits from QWidget, and all drawing will occur on the widget's surface.

233

The paintEvent() Method

For general purposes, painting is handled inside the `paintEvent()` function. Let's take a look at how to set up QPainter in the following code to draw a simple line:

```
def paintEvent(self, event):
    painter = QPainter() # Construct the painter
    painter.begin(self)
    painter.drawLine(260, 20, 260, 180)
    painter.end()
```

Drawing occurs between the `begin()` and `end()` methods on the paint device, referenced by `self`. The drawing is handled in between these two methods. Using `begin()` and `end()` is not required. You could construct a painter that takes as a parameter the paint device. However, `begin()` and `end()` can be used to catch any errors should the painter fail.

Other methods can also be called during the paint event. Since only one painter is allowed at a time, in the preceding example, we call different methods that all take the painter object as an argument.

```
self.drawPoints(painter)
self.drawLines(painter)
```

One of the settings that we can change in QPainter is the rendering quality using render hints. `QPainter.Antialiasing` creates smoother-looking curved edges.

```
painter.setRenderHint(QPainter.Antialiasing)
```

The QColor, QPen, and QBrush Classes

Some of the settings that can be modified include the color, width, and styles used to draw lines and shapes. The **QColor** class provides access to different color schemes, for example, RGB, HSV, and CMYK values. Colors can be specified by using either RGB hexadecimal strings, `'#112233'`; predefined color names, `Qt.blue` or `Qt.darkBlue`; or RGB values, `(233, 12, 43)`. QColor also includes an alpha channel used for giving colors transparency, where 0 is completely transparent and 255 is completely opaque. Listing 9-1 demonstrates all three of these types.

QPen is used for drawing lines and the outlines of shapes. The following code creates a black pen with a width of 2 pixels that draws dashed lines. The default style is `Qt.SolidLine`.

```
pen = QPen(QColor('#000000'), 2, Qt.DashLine)
painter.setPen(pen)
```

QBrush defines how to paint, that is, fill in, shapes. Brushes can have a color, a pattern, a gradient, or a texture. A magenta brush with the `Dense5Pattern` style is created with the following code. The default style is `Qt.SolidPattern`.

```
brush = QBrush(Qt.darkMagenta, Qt.Dense5Pattern)
painter.setBrush(brush)
```

If you wish to create multiple lines or shapes with different pens and brushes, make sure to call `setPen()` and/or `setBrush()` each time they need to be changed. Otherwise, QPainter will continue to use the pen and brush settings from the previous call.

Note   Calling `QPainter.begin()` will reset all the painter settings to default values.

Drawing Points and Lines

The `drawPoint()` method can be used to draw single pixels. By changing the width of the pen, you can draw wider points. The x and y values can either be explicitly defined or specified by using QPoint. An example of points is shown in Figure 9-2.

```
pen.setWidth(3)
painter.setPen(pen)
    painter.drawPoint(10, 15)
```

Note   The `drawPoint()` and other methods are specified using integer values. Some of the drawing methods allow you to also use floating-point values. Rather than import the `QPoint` and `QRect` classes, you should use `QPointF` and `QRectF`.

For drawing lines, there are the drawLine() or drawLines() methods. Each of the lines shown in Figure 9-2 displays different styles, widths, or colors. Lines are created by specifying a set of points, the starting x1 and y1 values and the ending x2 and y2 values.

```
pen.setStyle(Qt.DashLine) # Specify a style
painter.setPen(pen) # Set the pen
painter.drawLine(260, 20, 260, 180) # x1, y1, x2, y2
```
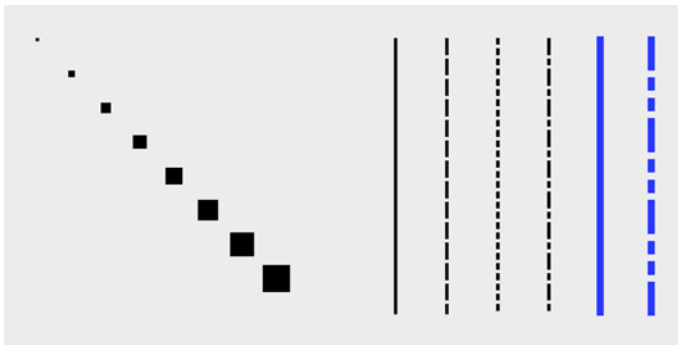

Figure 9-2. *Example of points and lines drawn using QPainter*

Drawing Text

The drawText() method is used to draw text on the paint device. An example of drawing text can be seen in Figure 9-3. We can make use of setFont() to apply different font settings.

```
painter.setFont(QFont("Helvetica", 15))
painter.setPen(pen)
painter.drawText(420, 110, text)
```

The text is drawn by first specifying the top-left coordinates on the paint device (think of text as being placed inside of a rectangle). This is the simplest way to draw text. For multiple lines or for wrapping text, use a QRect rectangle to contain the text.
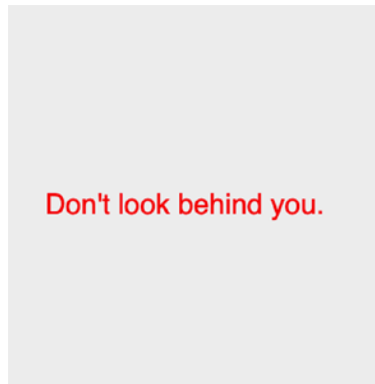
Figure 9-3. *A simple example of drawing text with QPainter*

Drawing Two-Dimensional Shapes

There are a few different ways to draw quadrilaterals using the drawRect() method. For this example, we will specify the top-left corner's coordinates followed by the width and height of the shape.

```
painter.drawRect(120, 220, 80, 80)
```

For each of the squares shown in the top-left corner of Figure 9-4, we begin by setting the pen and brush values before calling drawRect() to draw the shape. The first shape has a black pen with no brush; the second calls setBrush() to fill in the square. The next shape uses a red pen with a green brush. Finally, the last square shows an example of how to set the transparency of the pen object's color to 100.

```
blue_pen = QPen(QColor(32, 85, 230, 100), 5)
```

To draw irregular polygons, the QPolygon class can be used by specifying the point coordinates of each corner. The order that the points are created in the QPolygon object is the order in which they are drawn. The polygon object is then drawn using drawPolygon(). The polygon can be seen in the middle of the top row in Figure 9-4.

```
painter.drawPolygon(points)
```

237

QPainter can also draw rectangles with rounded corners. The process for drawing them is similar to drawing normal rectangles, except we need to specify the x and y radius values for the corners. Examples can be seen in Figure 9-4 in the top-right corner. The following snippet of code shows how to create a rounded rectangle by first creating the QRect rectangle and then specifying the style:

```
rect_1 = QRect(420, 340, 40, 60) # x, y, width, height
brush.setStyle(Qt.Dense1Pattern)
painter.setBrush(brush)
painter.drawRoundedRect(rect_1, 8, 8) # rect, x_rad, y_rad
```

For drawing abstract shapes, we need to use **QPainterPath**. Objects composed of different components, such as lines, rectangles, or curves, are called **painter paths**. An example of a painter path can be seen in the bottom-left corner of Figure 9-4.
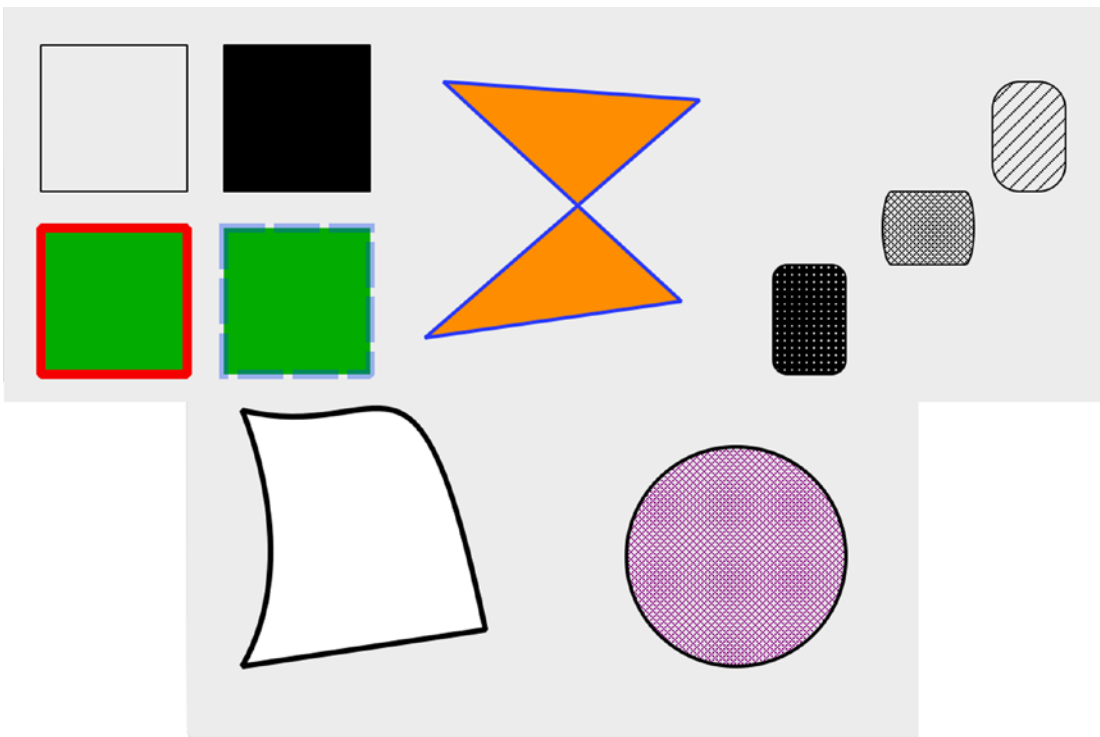
Figure 9-4. *Different shapes drawn with QPainter*

In the `drawCurves()` method of the earlier program, we first create a black `pen` and a white `brush` and an instance of `QPainterPath`. The `moveTo()` method moves to a position in the window without drawing any other components. We will start drawing at this position, `(30, 420)`.

```
path.cubicTo(30, 420, 65, 500, 30, 560)
path.lineTo(163, 540)
path.cubicTo(125, 360, 110, 440, 30, 420)
```

The `cubicTo()` method can be used to draw a parametric curve, also called a Bézier curve, from the starting position we moved to and the ending position, `(30, 560)`. The first two points, `(30, 420)` and `(65, 500)`, in `cubicTo()` are used to influence how the line curves between the starting and ending points. The next components of `path` are a line drawn with `lineTo()` and another curve. The abstract shape is closed with `closeSubpath()`, and the path is drawn using `drawPath()`.

The last shape we are going to look at is the ellipse which is drawn using `drawEllipse()`. For an ellipse, we need four values, the location of the center, and two radii values for the x and y directions. If the radii values are equal, we can draw a circle, like in the bottom-right corner of Figure 9-4. The following code shows how to draw an ellipse with a QPoint as the center coordinate, but the shape can also be drawn by defining a QRect.

```
painter.drawEllipse(QPoint(center_x, center_y), radius_x, radius_y)
```

Drawing Gradients

Gradients can be used along with QBrush to fill the inside of shapes. There are three different types of gradient styles in PyQt – linear, radial, and conical. For this example, we will use the **QLinearGradient** class to interpolate colors between two start and end points. The result can be seen in Figure 9-5.

The QLinearGradient constructor takes as arguments the area of the paint device where the gradient will occur, specified by the x1, y1, x2, y2 coordinates.

```
gradient = QLinearGradient(450, 480, 520, 550)
```

We can create points to start and stop painting colors using `setStopPoint()`. This method defines where one color ends and another color begins.
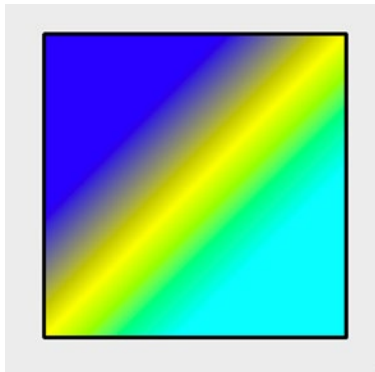
Figure 9-5. *Applying a gradient to a square*

Project 9.1 – Painter GUI

There are many digital art applications out there, filled to the brim with tools for drawing, painting, editing, and creating your own art on the computer. With QPainter, you could manually code each individual line and shape one by one. However, rather than going through that painstaking process to create digital works of art, the painter GUI project lays the foundation for creating your drawing application that could pave the way for a smoother drawing process. The interface can be seen in Figure 9-6.

For this first project, we will be looking to combine many of the concepts that you learned in previous chapters, including menubars, toolbars, status bars, dialog boxes, creating icons, and reimplementing event handlers, and combine them with the QPainter class. On top of it all, we will be sprinkling on a few new ideas, focusing on how to create tool tips and track the mouse's position.
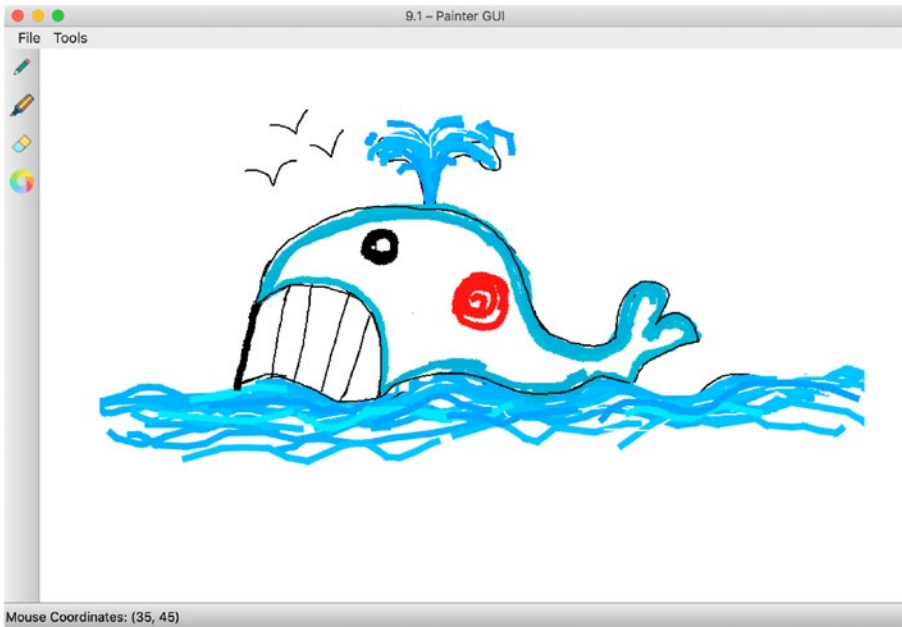
Figure 9-6. *The painter GUI with toolbar on the left side of the window and the mouse's current coordinates displayed in the status bar*

Painter GUI Solution

For the painter GUI in Listing 9-2, users will be able to draw using either a pencil or a marker tool, erase, and select colors using the QColorDialog. The items in the menu allow users to clear the current canvas, save their drawing, quit, and turn on or off antialiasing.

Listing 9-2. The code for creating the painter GUI

```
# painter.py
# Import necessary modules
import os, sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QAction, QLabel,
QToolBar, QStatusBar, QToolTip, QColorDialog, QFileDialog)
from PyQt5.QtGui import (QPainter, QPixmap, QPen, QColor, QIcon, QFont)
from PyQt5.QtCore import Qt, QSize, QPoint, QRect

# Create widget to be drawn on
class Canvas(QLabel):
```

241

```python
    def __init__(self, parent):
        super().__init__(parent)
        width, height = 900, 600

        self.parent = parent
        self.parent.setFixedSize(width, height)

        # Create pixmap object that will act as the canvas
        pixmap = QPixmap(width, height) # width, height
        pixmap.fill(Qt.white)
        self.setPixmap(pixmap)

        # Keep track of the mouse for getting mouse coordinates
        self.mouse_track_label = QLabel()
        self.setMouseTracking(True)

        # Initialize variables
        self.antialiasing_status = False
        self.eraser_selected = False

        self.last_mouse_pos = QPoint()
        self.drawing = False
        self.pen_color = Qt.black
        self.pen_width = 2

    def selectDrawingTool(self, tool):
        """
        Determine which tool in the toolbar has been selected.
        """
        if tool == "pencil":
            self.eraser_selected = False
            self.pen_width = 2
        elif tool == "marker":
            self.eraser_selected = False
            self.pen_width = 8
        elif tool == "eraser":
            self.eraser_selected = True
```

```python
        elif tool == "color":
            self.eraser_selected = False
            color = QColorDialog.getColor()
            if color.isValid():
                self.pen_color = color

    def mouseMoveEvent(self, event):
        """
        Handle mouse movements.
        Track coordinates of mouse in window and display in the status bar.
        """
        mouse_pos = event.pos()

        if (event.buttons() and Qt.LeftButton) and self.drawing:
            self.mouse_pos = event.pos()
            self.drawOnCanvas(mouse_pos)

        self.mouse_track_label.setVisible(True)
        sb_text = "Mouse Coordinates: ({}, {})".format(mouse_pos.x(),
        mouse_pos.y())
        self.mouse_track_label.setText(sb_text)
        self.parent.status_bar.addWidget(self.mouse_track_label)

    def drawOnCanvas(self, points):
        """
        Performs drawing on canvas.
        """
        painter = QPainter(self.pixmap())

        if self.antialiasing_status:
            painter.setRenderHint(QPainter.Antialiasing)

        if self.eraser_selected == False:
            pen = QPen(QColor(self.pen_color), self.pen_width)
            painter.setPen(pen)
            painter.drawLine(self.last_mouse_pos, points)

            # Update the mouse's position for next movement
            self.last_mouse_pos = points
```

243

```python
        elif self.eraser_selected == True:
            # Use the eraser
            eraser = QRect(points.x(), points.y(), 12, 12)
            painter.eraseRect(eraser)

        painter.end()
        self.update()

    def newCanvas(self):
        """
        Clears the current canvas.
        """
        self.pixmap().fill(Qt.white)
        self.update()

    def saveFile(self):
        """
        Save a .png image file of current pixmap area.
        """
        file_format = "png"
        default_name = os.path.curdir + "/untitled." + file_format
        file_name, _ = QFileDialog.getSaveFileName(self, "Save As",
                        default_name, "PNG Format (*.png)")

        if file_name:
            self.pixmap().save(file_name, file_format)

    def mousePressEvent(self, event):
        """
        Handle when mouse is pressed.
        """
        if event.button() == Qt.LeftButton:
            self.last_mouse_pos = event.pos()
            self.drawing = True

    def mouseReleaseEvent(self, event):
        """
        Handle when mouse is released.
```

```
        Check when eraser is no longer being used.
        """
        if event.button() == Qt.LeftButton:
            self.drawing = False
        elif self.eraser_selected == True:
            self.eraser_selected = False

    def paintEvent(self, event):
        """
        Create QPainter object.
        This is to prevent the chance of the painting being lost
        if the user changes windows.
        """
        painter = QPainter(self)

        target_rect = QRect()
        target_rect = event.rect()
        painter.drawPixmap(target_rect, self.pixmap(), target_rect)

class PainterWindow(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        #self.setMinimumSize(700, 600)
        self.setWindowTitle('9.1 - Painter GUI')

        QToolTip.setFont(QFont('Helvetica', 12))

        self.createCanvas()
        self.createMenu()
        self.createToolbar()

        self.show()
```

```python
def createCanvas(self):
    """
    Create the canvas object that inherits from QLabel.
    """
    self.canvas = Canvas(self)

    # Set the main window's central widget
    self.setCentralWidget(self.canvas)

def createMenu(self):
    """
    Set up the menu bar and status bar.
    """
    # Create file menu actions
    new_act = QAction('New Canvas', self)
    new_act.setShortcut('Ctrl+N')
    new_act.triggered.connect(self.canvas.newCanvas)

    save_file_act = QAction('Save File', self)
    save_file_act.setShortcut('Ctrl+S')
    save_file_act.triggered.connect(self.canvas.saveFile)

    quit_act = QAction("Quit", self)
    quit_act.setShortcut('Ctrl+Q')
    quit_act.triggered.connect(self.close)

    # Create tool menu actions
    anti_al_act = QAction('AntiAliasing', self, checkable=True)
    anti_al_act.triggered.connect(self.turnAntialiasingOn)

    # Create the menu bar
    menu_bar = self.menuBar()
    menu_bar.setNativeMenuBar(False)

    # Create file menu and add actions
    file_menu = menu_bar.addMenu('File')
    file_menu.addAction(new_act)
    file_menu.addAction(save_file_act)
```

```python
    file_menu.addSeparator()
    file_menu.addAction(quit_act)

    # Create tools menu and add actions
    file_menu = menu_bar.addMenu('Tools')
    file_menu.addAction(anti_al_act)

    self.status_bar = QStatusBar()
    self.setStatusBar(self.status_bar)

def createToolbar(self):
    """
    Create toolbar to contain painting tools.
    """
    tool_bar = QToolBar("Painting Toolbar")
    tool_bar.setIconSize(QSize(24, 24))
    # Set orientation of toolbar to the left side
    self.addToolBar(Qt.LeftToolBarArea, tool_bar)
    tool_bar.setMovable(False)

    # Create actions and tool tips and add them to the toolbar
    pencil_act = QAction(QIcon("icons/pencil.png"), 'Pencil', tool_bar)
    pencil_act.setToolTip('This is the <b>Pencil</b>.')
    pencil_act.triggered.connect(lambda: self.canvas.
    selectDrawingTool("pencil"))

    marker_act = QAction(QIcon("icons/marker.png"), 'Marker', tool_bar)
    marker_act.setToolTip('This is the <b>Marker</b>.')
    marker_act.triggered.connect(lambda: self.canvas.
    selectDrawingTool("marker"))

    eraser_act = QAction(QIcon("icons/eraser.png"), "Eraser", tool_bar)
    eraser_act.setToolTip('Use the <b>Eraser</b> to make it all
    disappear.')
    eraser_act.triggered.connect(lambda: self.canvas.
    selectDrawingTool("eraser"))
```

```
        color_act = QAction(QIcon("icons/colors.png"), "Colors", tool_bar)
        color_act.setToolTip('Choose a <b>Color</b> from the Color dialog.')
        color_act.triggered.connect(lambda: self.canvas.
        selectDrawingTool("color"))

        tool_bar.addAction(pencil_act)
        tool_bar.addAction(marker_act)
        tool_bar.addAction(eraser_act)
        tool_bar.addAction(color_act)

    def turnAntialiasingOn(self, state):
        """
        Turn antialiasing on or off.
        """
        if state:
            self.canvas.antialiasing_status = True
        else:
            self.canvas.antialiasing_status = False

    def leaveEvent(self, event):
        """
        QEvent class that is called when mouse leaves screen's space. Hide
        mouse coordinates in status bar if mouse leaves
        the window.
        """
        self.canvas.mouse_track_label.setVisible(False)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    app.setAttribute(Qt.AA_DontShowIconsInMenus, True)
    window = PainterWindow()
    sys.exit(app.exec_())
```

The final GUI can be seen in Figure 9-6.

Explanation

The painter GUI allows users to draw images on the canvas area. Unlike in the previous example where painting occurred on the main widget, for this example we will see how to subclass QLabel and reimplement its painting and mouse event handlers. The handling for some of the event handlers in this application was adapted from the Qt document web site.[1]

The program contains two classes – the Canvas class for drawing and the PainterWindow class for creating the menu and toolbar.

Creating the Canvas Class

Subclassing QLabel and reimplementing its paintEvent() method is a much easier way to manage drawing on a label object. We then create a pixmap and pass it to setPixmap(). Since QPixmap can be used as a QPaintDevice, using a pixmap makes handling the drawing and displaying of pixels much simpler. Also, using QPixmap means that we can set an initial background color using fill().

Next, we need to initialize a few variables and objects.

- mouse_track_label – A label for displaying the mouse's current position

- eraser_selected – True if the eraser is selected

- antialiasing_status – True if the user has checked the menu item for using antialiasing

- last_mouse_pos – Keep track of the mouse's last position when the left mouse button is pressed or when the mouse moves

- drawing – True if the left mouse button is pressed, indicating the user might be drawing

- pen_color, pen_width – Variables that hold the initial values of the pen and brush

---

[1]https://doc.qt.io/qt-5/qtwidgets-widgets-scribble-example.html

Since the user will use the mouse to draw in the GUI window, we need to handle the events when the mouse button is pressed or released and when the mouse is moved. We can use `setMouseTracking()` to keep track of the mouse cursor and return its coordinates in `mouseMoveEvent()`. Those coordinates are displayed in the status bar.

If the user presses the left mouse button while the cursor is in the window, we set `drawing` equal to `True` and store the current value of the mouse in `last_mouse_pos`. Then `drawOnCanvas()` is called in the `moveMouseEvent()`.

The user has four choices in the toolbar, including a pencil, marker, eraser, and a color selector. If a user selects a tool from the toolbar, a signal triggers the `selectDrawingTool()` slot, updating the current tool and painter settings.

The actual drawing is handled in `drawOnCanvas()`. An instance of `QPainter` is created that draws on the `pixmap`. We also check whether the `eraser_selected` is `True` or `False` to test whether we can draw or erase. The reimplementation of the `paintEvent()` creates a painter for the canvas area and draws the pixmap using `drawPixmap()`. By first drawing on a `QPixmap` in the `drawOnCanvas()` method and then copying the `QPixmap` onto the screen in the `paintEvent()`, we can ensure that our drawing won't be lost if the window is minimized.

The `Canvas` class also includes methods for clearing and saving the pixmap.

Creating the PainterWindow Class

The `PainterWindow` class creates the main menu, toolbar, tool tips for each of the buttons in the toolbar, and an instance of the `Canvas` class.

The File menu contains actions for clearing the canvas, saving the image, and quitting the application. The Tools menu contains a checkable menu item that turns antialiasing on or off.

The toolbar creates the actions and icons for the drawing tools. If a button is clicked, it triggers the `Canvas` class's `selectDrawingTool()` slot.

The reimplemented `leaveEvent()` handles if the mouse cursor moves outside the main window and sets the `mouse_track_label`'s visibility to `False`.

Handling Mouse Movement Events

This project displays the mouse's current x and y coordinates in the status bar. You may not want this kind of functionality, so the following code shows the basics for turning mouse tracking on and setting up `mouseMoveEvent()` to return the x and y values:

```
# Turn mouse tracking on
self.setMouseTracking(True)

    def mouseMoveEvent(self, event):
        mouse_pos = event.pos()
        pos_text = "Mouse Coordinates: ({}, {})".format(mouse_pos.x(),
        mouse_pos.y())
        print(pos_text)
```

Mouse move events occur whenever the mouse is moved, or when a mouse button is pressed or released.

Creating Tool Tips for Widgets

A user may often find themselves wondering about what some widget or action in a menu or toolbar actually does in an application. Tool tips are useful little bits of text that can be displayed to inform someone of a widget's function. Tool tips can be applied to any widget by using the setToolTip() method. Tips can display rich text formatted strings as shown in the following sample of code and in Figure 9-7. The font style and appearance of a tool tip can be adapted to fit your preferences.

```
eraser_act.setToolTip('Use the <b>Eraser</b> to make it all disappear.')
```
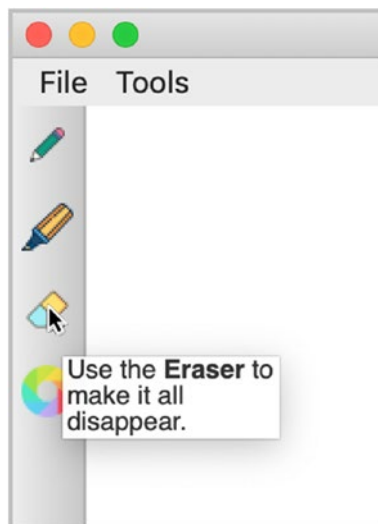

Figure 9-7. *The tool tip that is displayed when the user hovers over the eraser button*

Project 9.2 – Animation with QPropertyAnimation

The following project serves as an introduction to Qt's Graphics View Framework and the QAnimationProperty class. With the framework, applications can be created that allow users to interact with the items in the window.

A Graphics View is comprised of three components:

1. A **scene** created from the `QGraphicsScene` class. The scene creates the surface for managing 2D graphical items and must be created along with a view to visualize a scene.

2. `QGraphicsView` provides the **view** widget for visualizing the elements of a scene, creating a scroll area that allows user to navigate in the scene.

3. **Items** in the scene are based on the `QGraphicsItem` class. Users can interact with graphical items through mouse and key events, and drag and drop. Items also support collision detection.

QAnimationProperty is used to animate the properties of widgets and items. Animations in GUIs can be used for animating widgets. For example, you could animate a button that grows, shrinks, or rotates, or text that smoothly moves around in the window, or create widgets that fade in and out or change colors. QAnimationProperty only works with objects that inherit the **QObject** class. QObject is the base class for all objects created in PyQt.

Qt provides a number of simple items that inherit QGraphicsItem, including basic shapes, text, and pixmaps. These items already provide support for mouse and keyboard interaction. However, QGraphicsItem does not inherit QObject. Therefore, if you want to animate a graphics item with QPropertyAnimation, you must first create a new class that inherits from QObject and define new properties for the item.

Figure 9-8 shows an example of the scene we are going to create in this project.
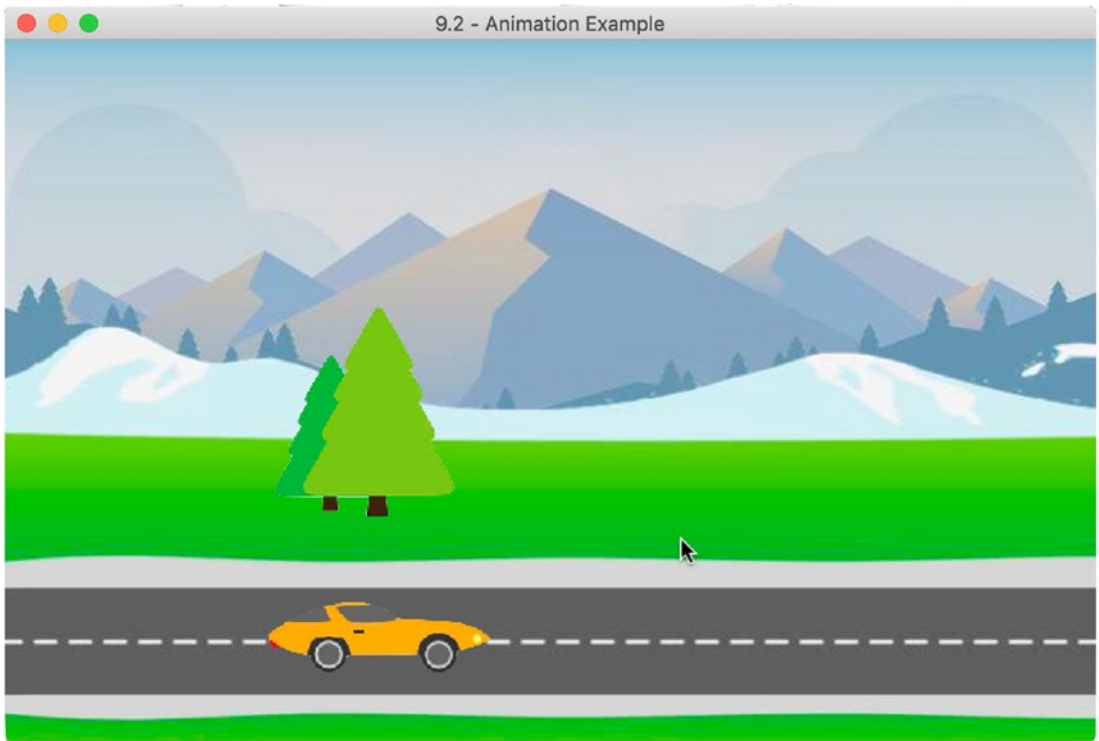
Figure 9-8. *The car and tree objects that move across the scene in the window*

Animation Solution

In the following application, you will find out how to create new properties for items using pyqtProperty, learn how to animate objects using the QPropertyAnimation class, and create a Qt Graphics View for displaying the items and animations. The code for creating simple animations can be found in Listing 9-3.

Listing 9-3. Code for animating objects in Qt's Graphics View Framework

```python
# animation.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QGraphicsView, QGraphicsScene,
QGraphicsPixmapItem)
from PyQt5.QtCore import (QObject, QPointF, QRectF,
    QPropertyAnimation, pyqtProperty)
from PyQt5.QtGui import QPixmap
```

```python
# Create Objects class that defines the position property of
# instances of the class using pyqtProperty.
class Objects(QObject):

    def __init__(self, image_path):
        super().__init__()

        item_pixmap = QPixmap(image_path)
        resize_item = item_pixmap.scaledToWidth(150)
        self.item = QGraphicsPixmapItem(resize_item)

    def _set_position(self, position):
        self.item.setPos(position)

    position = pyqtProperty(QPointF, fset=_set_position)

class AnimationScene(QGraphicsView):

    def __init__(self):
        super().__init__()
        self.initializeView()

    def initializeView(self):
        """
        Initialize the graphics view and display its contents
        to the screen.
        """
        self.setGeometry(100, 100, 700, 450)
        self.setWindowTitle('9.2 - Animation Example')

        self.createObjects()
        self.createScene()

        self.show()

    def createObjects(self):
        """
        Create instances of the Objects class, and set up the objects
        animations.
        """
```

```python
        # List that holds all of the animations.
        animations = []

        # Create the car object and car animation.
        self.car = Objects('images/car.png')

        self.car_anim = QPropertyAnimation(self.car, b"position")
        self.car_anim.setDuration(6000)

        self.car_anim.setStartValue(QPointF(-50, 350))
        self.car_anim.setKeyValueAt(0.3, QPointF(150, 350))
        self.car_anim.setKeyValueAt(0.6, QPointF(170, 350))
        self.car_anim.setEndValue(QPointF(750, 350))

        # Create the tree object and tree animation.
        self.tree = Objects('images/trees.png')

        self.tree_anim = QPropertyAnimation(self.tree, b"position")
        self.tree_anim.setDuration(6000)

        self.tree_anim.setStartValue(QPointF(750, 150))
        self.tree_anim.setKeyValueAt(0.3, QPointF(170, 150))
        self.tree_anim.setKeyValueAt(0.6, QPointF(150, 150))
        self.tree_anim.setEndValue(QPointF(-150, 150))

        # Add animations to the animations list, and start the
        # animations once the program begins running.
        animations.append(self.car_anim)
        animations.append(self.tree_anim)

        for anim in animations:
            anim.start()

    def createScene(self):
        """
        Create the graphics scene and add Objects instances
        to the scene.
        """
        self.scene = QGraphicsScene(self)
        self.scene.setSceneRect(0, 0, 700, 450)
```

```
        self.scene.addItem(self.car.item)
        self.scene.addItem(self.tree.item)
        self.setScene(self.scene)

    def drawBackground(self, painter, rect):
        """
        Reimplement QGraphicsView's drawBackground() method.
        """
        scene_rect = self.scene.sceneRect()

        background = QPixmap("images/highway.jpg")
        bg_rectf = QRectF(background.rect())
        painter.drawPixmap(scene_rect, background, bg_rectf)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = AnimationScene()
    sys.exit(app.exec_())
```

A still image from the animation project is shown in Figure 9-8.

Explanation

Since we are going to create a Graphics Scene, we need to import QGraphicsScene, QGraphicsView, and one of the QGraphicsItem classes. For this program, we import QGraphicsPixmapItem since we will be working with pixmaps. New Qt properties can be made using pyqtProperty.

Since QObject does not have a position property, we need to define one with pyqtProperty in the Objects class. QGraphicsPixmapItem() creates a pixmap that can be added into the QGraphicsScene. We create a position property that allows us to set and update the position of the object using fset. _set_position() passes the position to the QGraphicsItem.setPos() method, setting the position of the item to the coordinates specified by QPointF. Underscores in front of variable, method, or class names are used to denote private functions.

```
    def _set_position(self, position):
        self.item.setPos(position)
    position = pyqtProperty(QPointF, fset=_set_position)
```

The goal of this project is to animate two items, a car and a tree, in a QGraphicsScene. Let's first create the objects and the animations that will be placed into the scene. For this scene, the two items will move at the same time. Qt provides other classes for handling groups of animations, but for this example, the `QPropertyAnimation` and the `animations` list are used to keep track of the multiple animations.

Create the `car` item as an instance of the `Objects` class, and pass `car` and the `position` setter to `QPropertyAnimation()`. QPropertyAnimation will update the `position`'s value so that the `car` moves across the scene. To animate items, use `setDuration()` to set the amount of time the object moves in milliseconds, and specify start and end values of the property with `setStartValue()` and `setEndValue()`. The animation for the `car` is 6 seconds and starts off-screen on the left side and travels to the right. The tree is set up in a similar manner, but traveling in the opposite direction.

The `setKeyValueAt()` method allows us to create key frames at the given steps with the specified `QPointF` values. Using the key frames, the car and tree will appear to slow down as they pass in the scene. The `start()` method begins the animation.

Setting up a scene is simple. Create an instance of the `scene`, set the scene's size, add objects and their animations using `addItem()`, and then call `setScene()`.

Finally, a scene can be given a background using QBrush. If you want to use a background image, you will need to reimplement the QGraphicView's `drawBackground()` method like we do in this example.

Project 9.3 – RGB Slider Custom Widget

For this chapter's final project, we are going to take a look at making a custom, functional widget in PyQt. While PyQt offers a variety of widgets for building GUIs, every once in a while you might find yourself needing to create your own. One of the benefits of creating a customized widget is that you can either design a general widget that can be used by many different applications or create an application-specific widget that allows you to solve a specific problem.

There are quite a few techniques that you can use to create your own widgets, most which we have already seen in previous examples.

- Modifying the properties of PyQt's widgets by using built-in methods, such as `setAlignment()`, `setTextColor()`, and `setRange()`

- Creating style sheets to change a widget's existing behavior and appearances

- Subclassing widgets and reimplementing event handlers, or adding properties dynamically to QObject classes

- Creating composite widgets which are made up of two more types of widgets and arranged together using a layout

- Designing a completely new widget that subclasses QWidget and has its own unique properties and appearance

The RGB slider, shown in Figure 9-9, actually is created by combining a few of the preceding techniques listed. The widget uses Qt's QSlider and QSpinBox widgets for selecting RGB values and displays the color on a QLabel. The look of the sliders is modified by using style sheets. All of the widgets are then assembled into a parent widget which we can then import into other PyQt applications.

PyQt's Image Handling Classes

In previous examples, we have only worked with QPixmap for handling image data. Qt actually provides four different classes for working with images, each with their own special purposes.

QPixmap is the go-to choice for displaying images on the screen. Pixmaps can be used on QLabel widgets, or even on push buttons and other widgets that can display icons. **QImage** is optimized for reading, writing, and manipulating images, allowing direct access to an image's pixel data. QImage can also act as paint device.

Conversion between QImage and QPixmap is also possible. One possibility for using the two classes together is to load an image file with QImage, manipulate the image data, and then convert the image to a pixmap before displaying it on the screen. The RGB slider widget shows an example of how to convert between the two classes.

QBitmap is a subclass of QPixmap and provides monochrome (1-bit depth) pixmaps. **QPicture** is a paint device that replays QPainter commands, that is, you can create a picture from whatever device you are painting on. Pictures created with QPicture are resolution independent, appearing the same on any device.
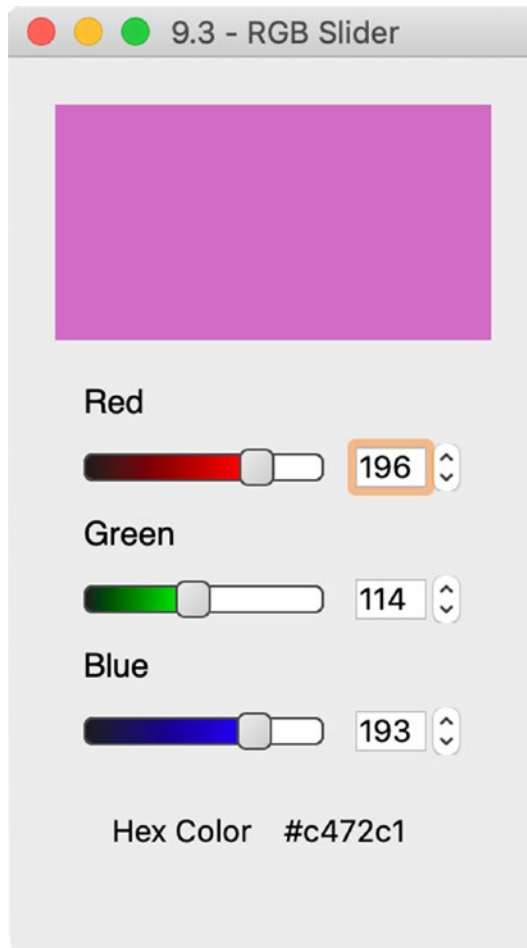
Figure 9-9. *A custom widget used to select colors using sliders and spin boxes*

Our custom widget uses two types of widgets for selecting RGB values – QSpinBox, which was introduced in Chapter 4, and a new widget, QSlider.

The QSlider Widget

The **QSlider** class provides a developer with a tool for selecting integer values within a bounded range. Sliders provide users with a convenient means for quickly selecting values or changing settings with only the movement of a simple handle. By default, sliders are arranged vertically, but that can be changed by passing `Qt.Horizontal` to the constructor.

In the following bit of code, you can see how to create an instance of QSlider, set the slider's maximum range value, and connect to valueChanged() to emit a signal when the slider's value has changed:

```
slider = QSlider(Qt.Horizontal, self)
# Default values are from 0 to 99
slider.setMaximum(200)
slider.valueChanged[int].connect(self.printSliderValue)

def printSliderValue(self, value):
    print(value)
```

An example of stylized slider widgets can be seen in Figure 9-9.

RGB Slider Solution

The RGB slider, which can be found in Listing 9-4, is a custom widget created by combining a few of Qt's built-in widgets – QLabel, QSlider, and QSpinBox. The appearance of the sliders is adjusted using style sheets so that they give visual feedback to the user about which RGB value they are adjusting. The sliders and spin boxes are connected together so that their values are in sync and so that the user can see the integer value on the RGB scale. The RGB values are also converted to hexadecimal format and displayed on the widget.

The sliders and spin boxes can be used to either find out the RGB or hexadecimal values for a color or use the reimplemented mousePressEvent() method so that a user can click a pixel in an image to find out its value. An example of this is shown in Listing 9-5, where you will also see how to import the RGB slider in a demo application.

Listing 9-4. Code for the RGB slider custom widget

```
# rgb_slider.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
    QSlider, QSpinBox, QHBoxLayout, QVBoxLayout, QGridLayout)
from PyQt5.QtGui import QImage, QPixmap, QColor, qRgb, QFont
from PyQt5.QtCore import Qt
```

```
style_sheet = """
    QSlider:groove:horizontal{
        border: 1px solid #000000;
        background: white;
        height: 10 px;
        border-radius: 4px
    }

    QSlider#Red:sub-page:horizontal{
        background: qlineargradient(x1: 1, y1: 0, x2: 0, y2: 1,
            stop: 0 #FF4242, stop: 1 #1C1C1C);
        background: qlineargradient(x1: 0, y1: 1, x2: 1, y2: 1,
            stop: 0 #1C1C1C, stop: 1 #FF0000);
        border: 1px solid #4C4B4B;
        height: 10px;
        border-radius: 4px;
    }

    QSlider::add-page:horizontal {
        background: #FFFFFF;
        border: 1px solid #4C4B4B;
        height: 10px;
        border-radius: 4px;
    }

    QSlider::handle:horizontal {
        background: qlineargradient(x1:0, y1:0, x2:1, y2:1,
            stop:0 #EEEEEE, stop:1 #CCCCCC);
        border: 1px solid #4C4B4B;
        width: 13px;
        margin-top: -3px;
        margin-bottom: -3px;
        border-radius: 4px;
    }

    QSlider::handle:horizontal:hover {
        background: qlineargradient(x1:0, y1:0, x2:1, y2:1,
            stop:0 #FFFFFF, stop:1 #DDDDDD);
```

```
        border: 1px solid #393838;
        border-radius: 4px;
    }

    QSlider#Green:sub-page:horizontal{
        background: qlineargradient(x1: 1, y1: 0, x2: 0, y2: 1,
            stop: 0 #FF4242, stop: 1 #1C1C1C);
        background: qlineargradient(x1: 0, y1: 1, x2: 1, y2: 1,
            stop: 0 #1C1C1C, stop: 1 #00FF00);
        border: 1px solid #4C4B4B;
        height: 10px;
        border-radius: 4px;
    }

    QSlider#Blue:sub-page:horizontal{
        background: qlineargradient(x1: 1, y1: 0, x2: 0, y2: 1,
            stop: 0 #FF4242, stop: 1 #1C1C1C);
        background: qlineargradient(x1: 0, y1: 1, x2: 1, y2: 1,
            stop: 0 #1C1C1C, stop: 1 #0000FF);
        border: 1px solid #4C4B4B;
        height: 10px;
        border-radius: 4px;
    }
"""

class RGBSlider(QWidget):

    def __init__(self, _image=None, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._image = _image

        self.initializeUI()


    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
```

```python
        self.setMinimumSize(225, 300)
        self.setWindowTitle('9.3 - RGB Slider')

        # Store the current pixel value
        self.current_val = QColor()

        self.setupWidgets()

        self.show()

    def setupWidgets(self):
        """
        Create instances of widgets and arrange them in layouts.
        """
        # Image that will display the current color set by
        # slider/spin_box values
        self.color_display = QImage(100, 100, QImage.Format_RGBX64)
        self.color_display.fill(Qt.black)

        self.cd_label = QLabel()
        self.cd_label.setPixmap(QPixmap.fromImage(self.color_display))
        self.cd_label.setScaledContents(True)

        # Create RGB sliders and spin boxes
        red_label = QLabel("Red")
        red_label.setFont(QFont('Helvetica', 14))
        self.red_slider = QSlider(Qt.Horizontal)
        self.red_slider.setObjectName("Red")
        self.red_slider.setMaximum(255)

        self.red_spinbox = QSpinBox()
        self.red_spinbox.setMaximum(255)

        green_label = QLabel("Green")
        green_label.setFont(QFont('Helvetica', 14))
        self.green_slider = QSlider(Qt.Horizontal)
        self.green_slider.setObjectName("Green")
        self.green_slider.setMaximum(255)
```

```python
        self.green_spinbox = QSpinBox()
        self.green_spinbox.setMaximum(255)

        blue_label = QLabel("Blue")
        blue_label.setFont(QFont('Helvetica', 14))
        self.blue_slider = QSlider(Qt.Horizontal)
        self.blue_slider.setObjectName("Blue")
        self.blue_slider.setMaximum(255)

        self.blue_spinbox = QSpinBox()
        self.blue_spinbox.setMaximum(255)

        # Use the hex labels to display color values in hex format
        hex_label = QLabel("Hex Color ")
        self.hex_values_label = QLabel()

        hex_h_box = QHBoxLayout()
        hex_h_box.addWidget(hex_label, Qt.AlignRight)
        hex_h_box.addWidget(self.hex_values_label, Qt.AlignRight)

        hex_container = QWidget()
        hex_container.setLayout(hex_h_box)

        # Create grid layout for sliders and spin boxes
        grid = QGridLayout()
        grid.addWidget(red_label, 0, 0, Qt.AlignLeft)
        grid.addWidget(self.red_slider, 1, 0)
        grid.addWidget(self.red_spinbox, 1, 1)
        grid.addWidget(green_label, 2, 0, Qt.AlignLeft)
        grid.addWidget(self.green_slider, 3, 0)
        grid.addWidget(self.green_spinbox, 3, 1)
        grid.addWidget(blue_label, 4, 0, Qt.AlignLeft)
        grid.addWidget(self.blue_slider, 5, 0)
        grid.addWidget(self.blue_spinbox, 5, 1)
        grid.addWidget(hex_container, 6, 0, 1, 0)

        # Use [] to pass arguments to the valueChanged signal
        # The sliders and spin boxes for each color should display the same
          values and be updated at the same time.
```

```
    self.red_slider.valueChanged['int'].connect(self.updateRedSpinBox)
    self.red_spinbox.valueChanged['int'].connect(self.updateRedSlider)

    self.green_slider.valueChanged['int'].connect(self.
    updateGreenSpinBox)
    self.green_spinbox.valueChanged['int'].connect(self.
    updateGreenSlider)

    self.blue_slider.valueChanged['int'].connect(self.
    updateBlueSpinBox)
    self.blue_spinbox.valueChanged['int'].connect(self.
    updateBlueSlider)

    # Create container for rgb widgets
    rgb_widgets = QWidget()
    rgb_widgets.setLayout(grid)

    v_box = QVBoxLayout()
    v_box.addWidget(self.cd_label)
    v_box.addWidget(rgb_widgets)

    self.setLayout(v_box)

# The following methods update the red, green, and blue
# sliders and spin boxes.
def updateRedSpinBox(self, value):
    self.red_spinbox.setValue(value)
    self.redValue(value)

def updateRedSlider(self, value):
    self.red_slider.setValue(value)
    self.redValue(value)

def updateGreenSpinBox(self, value):
    self.green_spinbox.setValue(value)
    self.greenValue(value)
```

```python
    def updateGreenSlider(self, value):
        self.green_slider.setValue(value)
        self.greenValue(value)

    def updateBlueSpinBox(self, value):
        self.blue_spinbox.setValue(value)
        self.blueValue(value)

    def updateBlueSlider(self, value):
        self.blue_slider.setValue(value)
        self.blueValue(value)

    # Create new colors based upon the changes to the RGB values
    def redValue(self, value):
        new_color = qRgb(value, self.current_val.green(), self.current_val.
        blue())
        self.updateColorInfo(new_color)

    def greenValue(self, value):
        new_color = qRgb(self.current_val.red(), value, self.current_val.
        blue())
        self.updateColorInfo(new_color)

    def blueValue(self, value):
        new_color = qRgb(self.current_val.red(), self.current_val.green(),
        value)
        self.updateColorInfo(new_color)

    def updateColorInfo(self, color):
        """
        Update color displayed in image and set the hex values accordingly.
        """
        self.current_val = QColor(color)
        self.color_display.fill(color)

        self.cd_label.setPixmap(QPixmap.fromImage(self.color_display))
        self.hex_values_label.setText("{}".format(self.current_val.name()))
```

```
 def getPixelValues(self, event):
        """

        The method reimplements the mousePressEvent method.
To use, set a widget's mousePressEvent equal to getPixelValues, like so:
            image_label.mousePressEvent = rgb_slider.getPixelValues
        If an _image != None, then the user can select pixels in the
        images, and update the sliders to get view the color, and get the
        rgb and hex values.
        """

        x = event.x()
        y = event.y()

        # valid() returns true if the point selected is a valid
        # coordinate pair within the image
        if self._image.valid(x, y):
            self.current_val = QColor(self._image.pixel(x, y))

            red_val = self.current_val.red()
            green_val = self.current_val.green()
            blue_val = self.current_val.blue()

            self.updateRedSpinBox(red_val)
            self.updateRedSlider(red_val)
            self.updateGreenSpinBox(green_val)
            self.updateGreenSlider(green_val)
            self.updateBlueSpinBox(blue_val)
            self.updateBlueSlider(blue_val)
```

An example of the stand-alone widget can be seen in Figure 9-9.

Explanation

We need to import quite a few classes. One worth noting, qRgb, is actually a typedef that creates an unsigned int representing the RGB value triplet (r, g, b).

The style sheet that follows the imports is used for changing the appearance of the sliders. We want to modify their appearance so that they give the user more feedback about which RGB values are being changed. Each slider is given an ID selector using the

setObjectName() method. If no ID selector is used in the style sheet, then that style is applied to all of the QSlider objects. The sliders use linear gradients so that users can get a visual representation of how much of the red, green, and blue colors are being used. Refer back to Chapter 6 for a refresher about style sheets.

The RGBSlider class inherits from QWidget. For this class, the user can pass an image and other arguments as parameters in the constructor.

```
class RGBSlider(QWidget):
    def __init__(self, _image=None, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self._image = _image
```

In setupWidgets(), a QImage object is created that will display the color created from the RGB values. To display the image in the widget, convert the QImage to a QPixmap using

```
self.cd_label.setPixmap(QPixmap.fromImage(self.color_display))
```

The contents of the label are then scaled to fit the window's size.

Next, we create each of the red, green, and blue QSlider and QSpinBox widgets and the labels for displaying the hexadecimal value. The sliders' maximum values are set to 255, since RGB values are in the range of 0–255. These widgets are then arranged using QGridLayout.

Updating the Sliders and Spin Boxes

QSlider and QSpinBox can both emit the valueChanged() signal. We can connect the sliders and spin boxes so that their values change relative to each other. For example, when the red_slider emits a signal, it triggers the updateRedSpinBox() slot, which then updates the red_spinbox value using setValue(). A similar process happens for the red_spinbox and for the green and blue sliders and spin boxes.

```
red_slider.valueChanged['int'].connect(self.updateRedSpinBox)
red_spinbox.valueChanged['int'].connect(self.updateRedSlider)

def updateRedSpinBox(self, value):
    self.red_spinbox.setValue(value)
    self.redValue(value)
```

These widgets are contained in the `rgb_widgets QWidget`. The last thing to do is to arrange the widgets in the main layout.

Updating the Colors

When a signal triggers a slot, it uses `value` to update the corresponding slider or spin box and then calls a function that will create a new color from the red, green, or blue values. The `redValue()` function shown in the following code creates a new `qRgb` color, using the new red `value` and the `current_val`'s `green()` and `blue()` colors. `current_val` is an instance of `QColor`. The QColor class has functions that we can use to access an image's RGB (or other color formats) value.

```
def redValue(self, value):
        new_color = qRgb(value, self.current_val.green(), self.current_val.
        blue())
        self.updateColorInfo(new_color)
```

The `new_color` is then passed to `updateColorInfo()`. Green and blue colors are handled in a similar fashion. Next we have to create a `QColor` from the `qRgb` value and store it in `current_val`. The `QImage color_display` is updated with `fill()`, which is then converted to a `QPixmap` and displayed on the `cd_label`.

The last thing to do is to update the hexadecimal labels using `QColor.name()`. This function returns the name of the color in the format "#RRGGBB".

Adding Methods to a Custom Widget

The options for methods that you could create for a custom widget are numerous. One option is to create methods that allow the user to modify the behavior or appearance of your custom widget. Another option is to use the event handlers to check for keyboard or mouse events that could be used to interact with your GUI.

`getPixelValue()` is a reimplementation of the `mousePressEvent()` event handler. If an image is passed into the `RGBSlider` constructor, then `_image` is not `None`, and the user can click points in the image to get their corresponding pixel values. `QColor.pixel()` gets a pixel's RGB values. Then, we update `current_val` to use the selected pixel's red, blue, and green values. These values are then passed back into the functions that will update the sliders, spin boxes, labels, and `QImage`.

The following example demonstrates how to implement the color selecting feature.

RGB Slider Demo

One reason for creating a custom widget is so that it can be used in other applications. The following program is a short example of how to import and set up the RGB slider shown in Project 9.3. For this example, an image is displayed in the window alongside the RGB slider. Users can click points within the image and see the RGB and hexadecimal values change in real time.

This short program's GUI can be seen in Figure 9-10.

Listing 9-5. Code that shows an example for using the RGB slider widget

```python
# rgb_demo.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
    QHBoxLayout)
from PyQt5.QtGui import QPixmap, QImage
from PyQt5.QtCore import Qt
from rgb_slider import RGBSlider, style_sheet

class ImageDemo(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(225, 300)
        self.setWindowTitle('9.3 - Custom Widget')

        # Load image
        image = QImage("images/chameleon.png")
```

```
        # Create instance of RGB slider widget and pass the image as an
          argument to RGBSlider
        rgb_slider = RGBSlider(image)

        image_label = QLabel()
        image_label.setAlignment(Qt.AlignTop)
        image_label.setPixmap(QPixmap().fromImage(image))
        # Reimplement the label's mousePressEvent
        image_label.mousePressEvent = rgb_slider.getPixelValues

        h_box = QHBoxLayout()
        h_box.addWidget(rgb_slider)
        h_box.addWidget(image_label)

        self.setLayout(h_box)
        self.show()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    # Use the style_sheet from rgb_slider
    app.setStyleSheet(style_sheet)
    window = ImageDemo()
    sys.exit(app.exec_())
```

The RGB slider is a general widget and can be imported into different types of programs. An example can be seen in Figure 9-10.
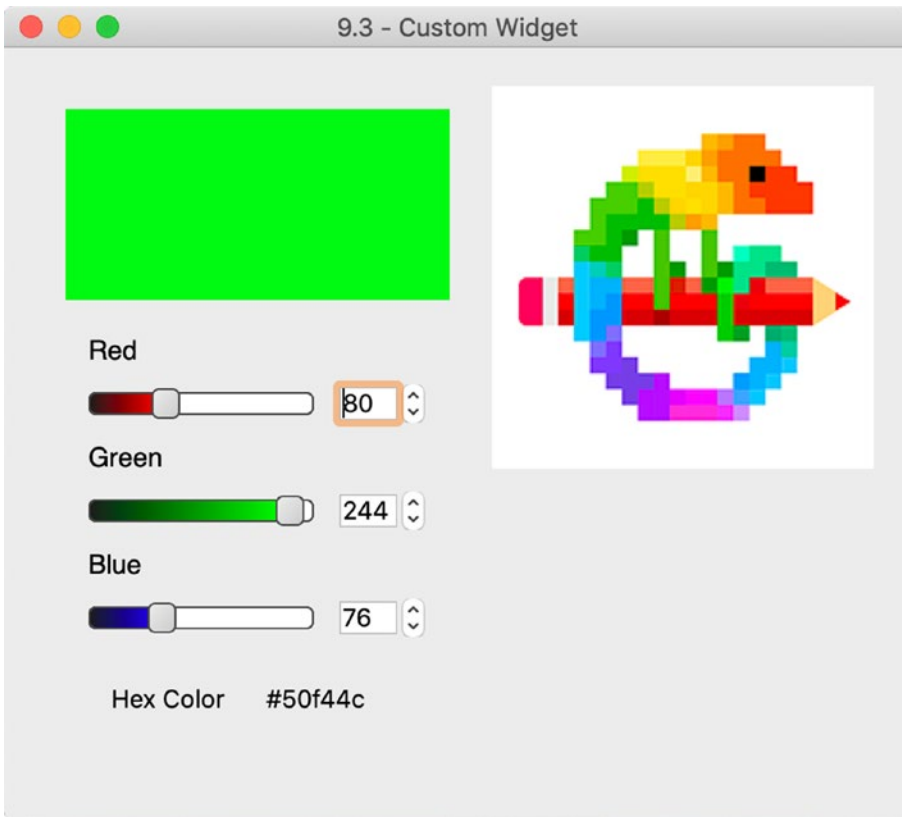
Figure 9-10. *An example of including the custom RGB slider in an application*

Explanation

Let's get started by importing the classes we need, including the RGB slider and style sheet from rgb_slider.py.

```
from rgb_slider import RGBSlider, style_sheet
```

In the ImageDemo class, set up the window, create an instance of the RGB slider, and load an image. For this application, we are still creating the image as an instance of QImage and then converting it to a QPixmap. QImage is used so that we can get access to the image's pixel information.

If you only want to use the slider to get different RGB or hexadecimal values, then the application is finished. Or you could add other functionality to the RGB slider to use in your own projects.

However, we could also reimplement the `QLabel` object's mouse event handler. When the mouse is clicked over a point in the label, we can use the x and y coordinates from the event to update the values in the RGB slider widget using the `RGBSlider` class's `getPixelValues()` method.

```
image_label.mousePressEvent = rgb_slider.getPixelValues
```

Summary

PyQt5's graphics and painting system is an extensive topic that could be an entire book by itself. The QPainter class is important for performing the painting on widgets and on other paint devices. QPainter works together with the QPaintEngine and QPaintDevice classes to provide the tools you need for creating two-dimensional drawing applications.

In Chapter 9, we have taken a look at some of QPainter's functions for drawing lines, primitive and abstract shapes. Together with QPen, QBrush, and QColor, QPainter is able to create some rather beautiful digital images. To materialize this concept, we created a simple painting application. Hopefully, you use that application and add even more drawing features.

We also saw how to create properties for objects made from the QObject class and then animate those objects in Qt Graphics View Framework. It is not covered in this book, but you could use the Graphics View to create a GUI with items that are interactive.

Finally, one of PyQt's strengths comes from being able to customize the built-in widgets or to create your own widget that can then be imported seamlessly into other applications.

In Chapter 10, we will learn about data handling using databases and PyQt.

Introduction to Handling Databases

Data is fundamental to the ways that modern business, communications, science, and even our personal lives are changing. The information we create from our online shopping, social media posts, search-engine queries, and location data is collected, managed, and analyzed and can be used for a number of reasons, including to track consumer patterns, to train artificial intelligence algorithms, or even to study the geographic distribution of particular events such as diseases.

Data analysis is an important process, and this chapter will have a look at working with structured data for GUI development. Data can be stored in many different formats, including textual, visual, and multimedia.

In order to analyze data, we need to organize it into structures that we can store and then access electronically through a computer system. Sometimes you may only be working with a small dataset consisting of one or two files. Other times, you may need to access certain portions of an entire database filled with private information. A **database** is an organized collection of multiple datasets.

We generally view the data from files and databases in tables. The rows and columns of a table typically work best for handling the style of data in data files. If we had a dataset of employees in a company, each row might represent an individual employee in the company, while each column depicts the different types of attributes for each employee, such as their age, salary, and employee ID number.

This chapter will focus only on using PyQt's table classes for displaying and manipulating data. We will see how to use tables for creating the foundation for a spreadsheet editor, for working with CSV files, and for working with the SQL database management language. Of course, there are also other formats for viewing data, namely, lists and trees, should they better fit your application's requirements.

In Chapter 10, we are going to take a look at creating GUIs that will

- Take a look at PyQt's convenience class for making tables, `QTableWidget`

- Find out how to add context menus to GUI applications

- Learn about Qt's model/view architecture for working with data using the `QTableView` class

- See an example of how to work with CSV files in PyQt

- Introduce the `QtSql` module for working with SQL and databases

The QTableWidget Class

The **QTableWidget** class provides a means to display and organize data in tabular form, presenting the information in rows and columns. Using tables breaks down data into a more quickly readable layout. An example of PyQt's tables can be seen in Figure 10-1.

QTableWidget provides you with the standard tools that you will need to create tables, including the ability to edit cells, set the number of rows and columns, and add vertical or horizontal header labels.

To create a QTableWidget object, you could pass the number of rows and columns as parameters to the QTableWidget, like in the following code:

```
table_widget = QTableWidget(10, 10, self)
```

Or you could construct a table using the `setRowCount()` and `setColumnCount()` methods.

```
table_widget = QTableWidget()
# Set initial row and column values
table_widget.setRowCount(10)
table_widget.setColumnCount(10)
```

You can also add items to the table programmatically using the `setItem()` method. This allows you to set the row and column values, and an item for the cell using **QTableWidgetItem**. In the following code, the item `Kalani` is inserted in row `0` and column `0`:

```
self.table_widget.setItem(0,0, QTableWidgetItem("Name"))
self.table_widget.setItem(1,0, QTableWidgetItem("Kalani"))
```

Setting either horizontal or vertical header labels is done with
`setHorizontalHeaderItem()` or `setHorizontalHeaderLabels()`. Change `Horizontal` to
`Vertical` for the vertical header.

For the first example in this chapter, Listing 10-1, we will be taking a look at how to
use QTableWidget to create the foundation for an application to edit spreadsheets and
how to use a context menu to manipulate the contents of the table widget.

Listing 10-1. Example code that uses the QTableWidget class and some of its
functions

```python
# spreadsheet.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow,
    QTableWidget, QTableWidgetItem, QMenu, QAction,
    QInputDialog)

class SpreadsheetFramework(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):
        self.setMinimumSize(1000, 500)
        self.setWindowTitle("Spreadsheet - Table Example")

        # Used for copy and paste actions
        self.item_text = None

        self.createMenu()
        self.createTable()

        self.show()

    def createTable(self):
        """
        Set up table widget.
```

```python
        """
        self.table_widget = QTableWidget()

        # Set initial row and column values
        self.table_widget.setRowCount(10)
        self.table_widget.setColumnCount(10)

        # Set focus on cell in the table
        self.table_widget.setCurrentCell(0, 0)

        # When the horizontal headers are double-clicked, emit a signal
        self.table_widget.horizontalHeader().sectionDoubleClicked.
        connect(self.changeHeader)

        self.setCentralWidget(self.table_widget)

    def createMenu(self):
        """
        Set up the menu bar.
        """
        # Create file menu actions
        quit_act = QAction("Quit", self)
        quit_act.setShortcut('Ctrl+Q')
        quit_act.triggered.connect(self.close)

        # Create table menu actions
        self.add_row_above_act = QAction("Add Row Above", self)
        self.add_row_above_act.triggered.connect(self.addRowAbove)

        self.add_row_below_act = QAction("Add Row Below", self)
        self.add_row_below_act.triggered.connect(self.addRowBelow)

        self.add_col_before_act = QAction("Add Column Before", self)
        self.add_col_before_act.triggered.connect(self.addColumnBefore)

        self.add_col_after_act = QAction("Add Column After", self)
        self.add_col_after_act.triggered.connect(self.addColumnAfter)

        self.delete_row_act = QAction("Delete Row", self)
        self.delete_row_act.triggered.connect(self.deleteRow)
```

```python
    self.delete_col_act = QAction("Delete Column", self)
    self.delete_col_act.triggered.connect(self.deleteColumn)

    self.clear_table_act = QAction("Clear All", self)
    self.clear_table_act.triggered.connect(self.clearTable)

    # Create the menu bar
    menu_bar = self.menuBar()
    menu_bar.setNativeMenuBar(False)

    # Create file menu and add actions
    file_menu = menu_bar.addMenu('File')
    file_menu.addAction(quit_act)

    # Create table menu and add actions
    table_menu = menu_bar.addMenu('Table')
    table_menu.addAction(self.add_row_above_act)
    table_menu.addAction(self.add_row_below_act)
    table_menu.addSeparator()
    table_menu.addAction(self.add_col_before_act)
    table_menu.addAction(self.add_col_after_act)
    table_menu.addSeparator()
    table_menu.addAction(self.delete_row_act)
    table_menu.addAction(self.delete_col_act)
    table_menu.addSeparator()
    table_menu.addAction(self.clear_table_act)

def contextMenuEvent(self, event):
    """
    Create context menu and actions.
    """
    context_menu = QMenu(self)

    context_menu.addAction(self.add_row_above_act)
    context_menu.addAction(self.add_row_below_act)
    context_menu.addSeparator()
    context_menu.addAction(self.add_col_before_act)
    context_menu.addAction(self.add_col_after_act)
```

279

```python
        context_menu.addSeparator()
        context_menu.addAction(self.delete_row_act)
        context_menu.addAction(self.delete_col_act)
        context_menu.addSeparator()
        copy_act = context_menu.addAction("Copy")
        paste_act = context_menu.addAction("Paste")
        context_menu.addSeparator()
        context_menu.addAction(self.clear_table_act)

        # Execute the context_menu and return the action selected.
          mapToGlobal() translates the position of the window coordinates to
          the global screen coordinates. This way we can detect if a right-
          click occurred inside of the GUI and display the context menu.
        action = context_menu.exec_(self.mapToGlobal(event.pos()))

        # To check for actions selected in the context menu that were not
          created in the menu bar.
        if action == copy_act:
            self.copyItem()
        if action == paste_act:
            self.pasteItem()

    def changeHeader(self):
        """
        Change horizontal headers by returning the text from input dialog.
        """
        col = self.table_widget.currentColumn()

        text, ok = QInputDialog.getText(self, "Enter Header", "Header text:")

        if ok and text != "":
            self.table_widget.setHorizontalHeaderItem(col,
            QTableWidgetItem(text))
        else:
            pass
```

```python
    def copyItem(self):
        """
        If the current cell selected is not empty, store the text.
        """
        if self.table_widget.currentItem() != None:
            self.item_text = self.table_widget.currentItem().text()

    def pasteItem(self):
        """
        Set item for selected cell.
        """
        if self.item_text != None:
            row = self.table_widget.currentRow()
            column = self.table_widget.currentColumn()
            self.table_widget.setItem(row, column, QTableWidgetItem(self.
            item_text))

    def addRowAbove(self):
        current_row = self.table_widget.currentRow()
        self.table_widget.insertRow(current_row)

    def addRowBelow(self):
        current_row = self.table_widget.currentRow()
        self.table_widget.insertRow(current_row + 1)

    def addColumnBefore(self):
        current_col = self.table_widget.currentColumn()
        self.table_widget.insertColumn(current_col)

    def addColumnAfter(self):
        current_col = self.table_widget.currentColumn()
        self.table_widget.insertColumn(current_col + 1)

    def deleteRow(self):
        current_row = self.table_widget.currentRow()
        self.table_widget.removeRow(current_row)

    def deleteColumn(self):
        current_col = self.table_widget.currentColumn()
        self.table_widget.removeColumn(current_col)
```

```
    def clearTable(self):
        self.table_widget.clear()

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = SpreadsheetFramework()
    sys.exit(app.exec_())
```

Figure 10-1 displays the GUI for this application, including the QTableWidget with examples of data already entered into some of the rows and columns, and horizontal headers.

Figure 10-1. *Example of a table from the QTableWidget class*

Explanation

When we import classes in the beginning of the program, we need to make sure to include `QTableWidget` and `QTableWidgetItem`, which is used to create items for the table widget. A table is composed of a group of cells, and the items are the bits of textual information in each one. QTableWidget has a number of signals for checking to see if cells or items have been clicked, double-clicked, or even altered.

Next, create the menubar with `File` and `Table` menus. QTableWidget includes a few methods for manipulating table objects. The `Table` menu creates actions that put those methods to use. These actions include

- Adding rows above or below the currently selected row using `insertRow()`

- Adding columns before or after the currently selected column using `insertColumn()`

- Deleting the current row or column using `removeRow()` or `removeColumn()`

- Clearing the entire table, including items and headers with `clear()`

Since we are working with a table, if we are going to manipulate the rows or columns, we first need to know which row or column is currently selected. For example, when `add_row_above_act` is clicked, it triggers a signal that calls `addRowAbove()`. We first find out the row that is selected using `currentRow()`.

```
current_row = self.table_widget.currentRow()
self.table_widget.insertRow(current_row)
```

A new row is then inserted in the current row's location, causing all other rows to move down. For methods that manipulate columns, use the `currentColumn()` method.

Changing header labels in QTableWidget can either be done directly in code or by using a slightly indirect approach. Headers for tables are created using **QHeaderView** in the QTableView class (which we will cover later in this chapter's project). Since QTableWidget inherits from the QTableView class, we also have access to its functions. In the following line of code, we are able to obtain the QHeaderView object using `table_widget.horizontalHeader()`. From there, we can connect to the QHeaderView signal `sectionDoubleClicked()`, checking to see if the user double-clicked a header section. If they did, a signal triggers the `changeHeader()` method.

```
self.table_widget.horizontalHeader().sectionDoubleClicked.connect(self.
changeHeader)
```

From there, we get the column for the current header and show a `QInputDialog` to get the header label from the user. Finally, the item for the horizontal header is set using `setHorizontalHeaderItem()`.

Creating Context Menus

This application also introduces how to create a context menu, sometimes called a pop-up menu, that appears in the window due to a user's interaction, such as when the right mouse button is clicked. A **context menu** displays a list of commands, such as Back Page or Reload Page, that make interacting with the GUI even more convenient. Context menus can also be set for managing specific widgets.

Since context menus are caused by events, we can reimplement the contextMenuEvent().

```
def contextMenuEvent(self, event):
    context_menu = QMenu(self)
    context_menu.addAction(self.add_row_above_act)
```

A context menu is typically created using QMenu(). You can either use existing actions that are created in the menubar or the toolbar, or you can create new ones. In the preceding example, two actions are created specifically for the context menu, copy_act and paste_act. If a cell in the table is not empty, we "copy" the text to item_text. In the pasteItem() slot, the current row and column of the selected cell is checked. We then "paste" the item using setItem(). The copy and paste actions could also be implemented using the QClipboard.

The context menu is displayed using exec_(). We pass self.mapToGlobal() as an argument to get the coordinates of the mouse relative to the screen. An example of the context menu can be seen in Figure 10-2.
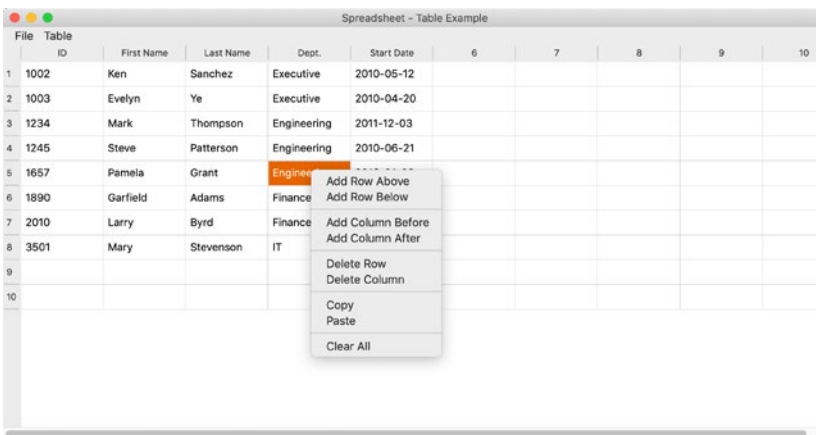
Figure 10-2. Example of a context menu that displays actions for editing the table widget

The QTableWidget is actually a **convenience class**, providing simplified access to other classes, namely, QTableView and QAbstractModel. Before learning about accessing databases with PyQt, you should take a moment to get familiar with the model/view architecture used by Qt.

Introduction to Model/View Programming

Qt, and therefore PyQt, needs a system to access, display, and manage data that can be presented to the user. An older technique used for managing the relationship between data and its visual representation for user interfaces is the **model-view-controller** (**MVC**) software design pattern. MVC divides a program's logic into three interlinked components – a model, a view, and a controller.

PyQt utilizes a similar design pattern that is based on MVC – the model/view architecture.

The Components of the Model/View Architecture

Model/view programming also separates the logic between three components, but combines the view and the controller objects, and introduces a new element – a delegate. A diagram of the architecture can be seen in Figure 10-3.

- Model – The class that communicates with the data source, accessing the data, and provides a point of connection between the data and the view and delegate.

- View – The class that is responsible for displaying the data to the user, either in list, table, or tree formats, and for retrieving items of data from the model using model indexes. The view also has similar functionality to the **controller** in the MVC pattern, which handles the input from a user's interaction with items displayed in the view.

- Delegate – The class that is in charge of painting items and providing editors in the view. The delegate also communicates back to the model if an item has been edited.

Using the model/view structure has quite a few benefits, specifically being ideal for developing large-scale applications, giving more flexibility and control over the appearance and editing of data items, simplifying the framework for displaying data, and offering the ability to display multiple views of a model at the same time.

285
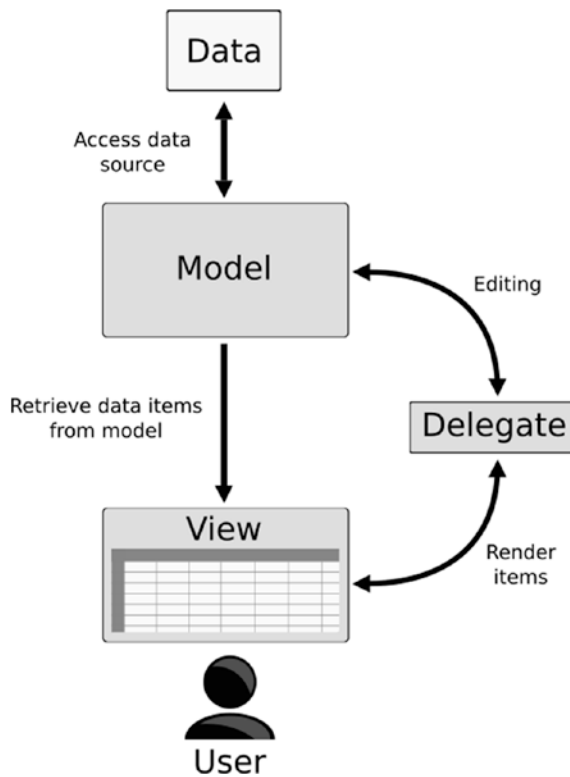
Figure 10-3. *The model accesses data from the data source and provides data to the view. The view presents items stored in a model and reflects changes to the data in the model. The delegate is responsible for drawing items in the view and for handling the editing of the data in the model. (Adapted from* `https://doc.qt.io/` *web site)*

PyQt's Model/View Classes

QTableWidget is one of a few convenience classes that PyQt provides for working with data. QTableWidget creates a table of items, QListWidget displays a list of items, and **QTreeWidget** provides a hierarchal treelike structure. An example of QListWidget can be seen in Chapter 8. These widgets provide all the tools necessary to work with data, and the view, model, and delegate classes all grouped into one class. However, these classes are more focused on item-based interfaces and are less flexible than working with the model/view structure. Each of these widgets inherits behavior from an abstract class, **QAbstractItemView**, creating the behavior for selecting items and managing headers.

An **abstract class** provides the points of connection, referred to as an **interface**, between other components, providing functionality and default implementation of features. Abstract classes can also be used to create custom models, views, or delegates.

- Models – All models are based on the `QAbstractItemModel` class, defining the interface used by both views and delegates to access data, and can be used to handle lists, tables, or trees. Data can take on a number of forms, including Python data structures, separate classes, files, or databases. Some other model classes are `QStandardItemModel`, `QFileSystemModel`, and SQL-related models.

- Views – All views are based on `QAbstractItemView` and are used to display data items from a data source, including `QListView`, `QTableView`, and `QTreeView`.

- Delegates – The base class is `QAbstractItemDelegate`, responsible for drawing items from the model and providing an editor widget for modifying items. For example, while editing a cell in a table, the editor widget, such as QLineEdit, is placed directly on top of the item.

The following example in Listing 10-2 demonstrates how to use the model/view classes for displaying data using tables. Chapter 12 contains an extra example that shows how to use QFileSystemModel and QTreeView to display the contents of directories on your computer.

Communication between the models, views, and delegates is handled by signals and slots. The model uses signals to notify the view about changes to the data. The view generates signals that provide information about how a user interacts with items. Signals from the delegate are emitted while editing an item to inform the model and view about the state of the editor.

The following program illustrates how to use model/view programming to display the contents of a small CSV file in a table view.

Listing 10-2. Code demonstrating how to design a GUI using model/view architecture

```
# model_view_ex.py
# Import necessary modules
import sys, csv
from PyQt5.QtWidgets import (QApplication, QWidget, QTableView, QVBoxLayout)
```

```python
from PyQt5.QtGui import QStandardItemModel, QStandardItem

class DisplayParts(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 450, 300)
        self.setWindowTitle('Model and View Example')

        self.setupModelView()

        self.show()

    def setupModelView(self):
        """
        Set up standard item model and table view.
        """
        self.model = QStandardItemModel()

        table_view = QTableView()
        # From QAbstractItemView.ExtendedSelection = 3
        table_view.SelectionMode(3)
        table_view.setModel(self.model)

        # Set initial row and column values
        self.model.setRowCount(3)
        self.model.setColumnCount(4)

        self.loadCSVFile()

        v_box = QVBoxLayout()
        v_box.addWidget(table_view)

        self.setLayout(v_box)
```
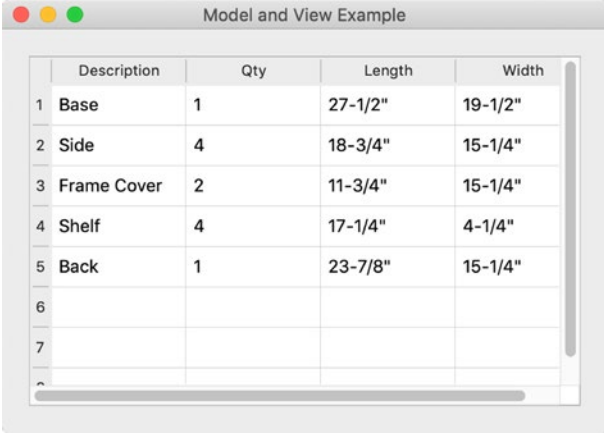
```python
    def loadCSVFile(self):
        """
        Load header and rows from CSV file.
        Items are constructed before adding them to the table.
        """
        file_name = "files/parts.csv"

        with open(file_name, "r") as csv_f:
            reader = csv.reader(csv_f)
            header_labels = next(reader)
            self.model.setHorizontalHeaderLabels(header_labels)
            for i, row in enumerate(csv.reader(csv_f)):
                items = [QStandardItem(item) for item in row]
                self.model.insertRow(i, items)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = DisplayParts()
    sys.exit(app.exec_())
```

The simple GUI created using model/view programming can be seen in Figure 10-4.

Figure 10-4. *Table created using the model/view architecture*

Explanation

The preceding example displays the contents of a CSV file in a table view and demonstrates how simple it is to use the model/view paradigm. Tables can be used to organize and display various types of data, such as employee or inventory information.

We begin by importing classes, including `QTableView` from the `QtWidgets` module, and the `QStandardItemModel` and `QStandardItem` classes from `QtGui`. QStandardItemModel will supply the item-based model we need to work with the data; QStandardItem provides the items that are used in the model.

Instances of both the model using `QStandardItemModel` as well as the `QTableView` class are created. There are different ways that users can select items in the table view. `SelectionMode()` handles how the view responds to users' selections. `ExtendedSelection` allows a user to select multiple items by pressing the Ctrl key (Cmd on MacOS) while clicking an item in the view or to select several items using the Shift key. To set up the view to display items in the model, you simply need to call the `setModel()` method.

```
table_view.setModel(self.model)
```

In the previous example where we looked at QTableWidget, the `setRowCount()` and `setColumnCount()` methods were called on the table widget. When using QTableView, these methods are not built-in and instead are called on the model.
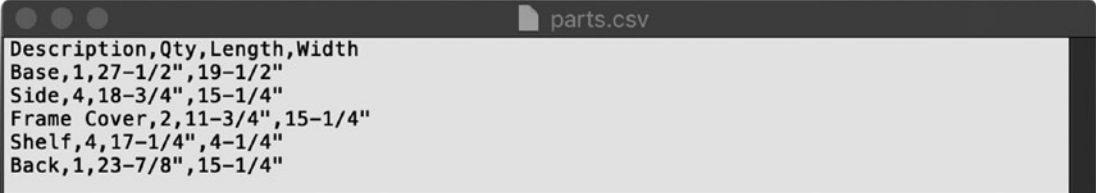
```
self.model.setRowCount(3)
```

Next, we call `loadCSVFile()` to read the contents of the data file and add the items to the model to be displayed in the view. The `table_view` widget is added to the `QVBoxLayout`.

In the `loadCSVLayout()` method, we can see how to read headers and data from a CSV file. Comma-separated values (CSV) is a very common format used for storing the data of spreadsheets and datasets. We open the file, set up the `reader` to read the sequences in the file, get the headers, and skip to the next line. For this example, we assume that the CSV file will have header labels. The horizontal labels of the model are set using the list of items from the first row.

```
self.model.setHorizontalHeaderLabels(header_labels)
```

For the remaining rows, we use a list comprehension to read the items for each row into a list and use `insertRow()` to insert the list of `items` into the `i`th row. Figure 10-5 shows the contents of the `parts.csv` file.


```
Description,Qty,Length,Width
Base,1,27-1/2",19-1/2"
Side,4,18-3/4",15-1/4"
Frame Cover,2,11-3/4",15-1/4"
Shelf,4,17-1/4",4-1/4"
Back,1,23-7/8",15-1/4"
```

Figure 10-5. *Example of the data stored in a CSV file*

Working with SQL Databases in PyQt

Now that we have looked at PyQt's model/view architecture and the QTableView class, let's move on and begin taking a look at how to use SQL for handling structured data.

What Is SQL?

The **Structured Query Language** (**SQL**) is a programming language designed for communication with databases. The data stored in databases is organized into a set of tables. The rows of the tables are referred to as **records**, and the columns are referred to as **fields**. Each column can only store a specific kind of information, such as names, dates, or numbers.

With SQL, we can **query** the data stored in **relational databases** – a collection of data items that have predefined relationships across multiple tables, marked by a unique identifier known as a **foreign key**. In a relational database, multiple tables comprise a **schema**, more than one schema makes up a database, and those databases are stored on a server. Relational databases allow for multiple users to handle the data at the same time. For this reason, accessing a database often requires a user to log in with a username and password in order to connect to the database.

This section will focus solely on using SQL along with classes from PyQt's QtSql module for creating a very basic database management system interface.

Working with Database Management Systems

The QtSql module provides drivers for a number of **relational database management systems** (**RDBMS**), including MySQL, Oracle, Microsoft SQL Server, PostgreSQL, and SQLite versions 2 and 3. An RDBMS is the software that allows users to interact with relational databases using SQL.

For the following examples, we will be using SQLite 3 since the library already comes shipped with Python and is included with Qt. SQLite is not a client-server database engine, so we do not need a database server. SQLite operates on a single file and is mainly used for small desktop applications.

Getting Familiar with SQL Commands

SQL already has its own commands for generating queries from databases. Using these commands, a user can perform a number of different actions for interacting with database tables. For example, the SQL SELECT statement can be used to retrieve records from a table. If you had a database for a dog identification registry that contained a table called dog_registry, you could select all of the records in the table with the following statement:

```
SELECT * FROM dog_registry
```

When you are creating a query, you should consider where you are getting your data from, including which database or table. You should keep in mind what fields you will use. And be mindful of any conditions in the selection. For example, do you need to display all the pets in the database, or only a specific breed of dog?

```
SELECT name FROM dog_registry WHERE breed = 'shiba inu'
```

Using different drivers will more than likely entail using different SQL syntax, but PyQt can handle the differences. The following table lists a few common SQLite 3 commands that will be used in this chapter's examples.

Table 10-1. *A list of common SQLite keywords and functions that can be found in this chapter*[1]

SQLite Keywords	Description
AUTOINCREMENT	Generates a unique number automatically when a new record is inserted into the table.
CREATE TABLE	Creates a new table in the database.
DELETE	Deletes a row from the table.
DROP TABLE	Deletes a table that already exists in the database.
FOREIGN KEY	Constraint that links two tables together.
FROM	Specifies the table to interact with when selecting or deleting data.
INTEGER	Signed integer data type.
INSERT INTO	Inserts new rows into the table.
MAX()	Function that finds the maximum value of a specified column.
NOT NULL	Constraint that ensures a column will not accept NULL values.
PRIMARY KEY	Constraint that uniquely identifies a record in the table.
REFERENCES	Used with FOREIGN KEY to specify another table which has relation with the first table.
SELECT	Selects data from a database.
SET	Identifies which columns and values should be updated.
UNIQUE	Constraint that ensures all values in a column are unique.
UPDATE	Updates existing values in a row.
VALUES	Defines the values of an INSERT INTO statement.
VARCHAR	Variable character data type for strings.
WHERE	Filters the results of a query to include only records that satisfy specific conditions.

In the following sections, we will see how to create a user interface that can be used to view a database's information in a table view.

---

[1]A full list of SQLite keywords can be found at `www.sqlite.org/lang_keywords.html`.

Project 10.1 – Account Management GUI

For this project, we are going to take a different approach to designing the account management GUI. This section builds up to the final project by working through a number of smaller example programs. There is a good deal of information to unpack, and if this is your first time working with SQL, especially to build an interface in PyQt, then the process for working with databases can become a little unclear.

Imagine you have a business and you want to create a database to keep track of your employees' information. You want to include information such as their first and last names, employee IDs, e-mail addresses, departments, and the countries where they work. (This could be extended to include more information such as salaries, phone numbers, and dates of hire.) In the beginning, a small database is okay. However, as your workforce builds, so will the information. Some employees may have the same first or last name, or work in the same country. You need a way to manage all of those employees so that fields in the database are populated with the correct information and data types.

Using a relational database, we can avoid issues with the data's integrity. We could set up multiple tables, one for the different employees' `accounts` and one for the `countries`. For this example, we only use repeating country names to demonstrate how to use PyQt's classes for working with relational databases. Figure 10-6 displays the account management GUI.

The project is broken down into the following parts:

1. Introduce how to use `QSqlDatabase` to connect to databases and `QSqlQuery` for creating queries

2. A few examples of how to use `QSqlQuery` for working with databases

3. Introduce `QSqlTableModel` for working with databases with no foreign keys

4. Show how use to `QSqlRelationalTableModel` to create tables with foreign key support

5. Create the account management GUI

Figure 10-6. *The account management GUI. The last row of the table displays a new record being added to the database*

Working with QtSql

In this first example, we are going to see how to use QSqlQuery to create a small database that we will be able to view in the account management GUI. The database has two tables, accounts and countries. The two tables are linked together through the country_id field in accounts and the id field in countries.

Listing 10-3. Code showing examples of how to create queries with QSqlQuery

```
# create_database.py
# Import necessary modules
import sys, random
from PyQt5.QtSql import QSqlDatabase, QSqlQuery

class CreateEmployeeData:
    """
    Create sample database for project.
```

```
    Class demonstrates how to connect to a database, create queries, and
    create tables and records in those tables.
    """
    # Create connection to database. If db file does not exist,
    # a new db file will be created.
    database = QSqlDatabase.addDatabase("QSQLITE") # SQLite version 3
    database.setDatabaseName("files/accounts.db")

    if not database.open():
        print("Unable to open data source file.")
        sys.exit(1) # Error code 1 - signifies error

    query = QSqlQuery()
    # Erase database contents so that we don't have duplicates
    query.exec_("DROP TABLE accounts")
    query.exec_("DROP TABLE countries")

    # Create accounts table
    query.exec_("""CREATE TABLE accounts (
                id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
                employee_id INTEGER NOT NULL,
                first_name VARCHAR(30) NOT NULL,
                last_name VARCHAR(30) NOT NULL,
                email VARCHAR(40) NOT NULL,
                department VARCHAR(20) NOT NULL,
                country_id VARCHAR(20) REFERENCES countries(id))""")

    # Positional binding to insert records into the database
    query.prepare("""INSERT INTO accounts (
                  employee_id, first_name, last_name,
                  email, department, country_id)
                  VALUES (?, ?, ?, ?, ?, ?)""")

    first_names = ["Emma", "Olivia", "Ava", "Isabella", "Sophia", "Mia",
    "Charlotte", "Amelia", "Evelyn", "Abigail", "Valorie", "Teesha",
    "Jazzmin", "Liam", "Noah", "William", "James", "Logan", "Benjamin",
    "Mason", "Elijah", "Oliver", "Jason", "Lucas", "Michael"]
```

```python
last_names = ["Smith", "Johnson", "Williams", "Brown", "Jones",
"Garcia", "Miller", "Davis", "Rodriguez", "Martinez", "Hernandez",
"Lopez", "Gonzalez", "Wilson", "Anderson", "Thomas", "Taylor", "Moore",
"Jackson", "Martin", "Lee", "Perez", "Thompson", "White", "Harris"]

employee_ids = random.sample(range(1000, 2500), len(first_names))

countries = {"USA": 1, "India": 2, "China": 3, "France": 4, "Germany": 5}
country_names = list(countries.keys())
country_codes = list(countries.values())

departments = ["Production", "R&D", "Marketing", "HR",
               "Finance", "Engineering", "Managerial"]

# Add the values to the query to be inserted in accounts
for f_name in first_names:
    l_name = last_names.pop()
    email = (l_name + f_name[0]).lower() + "@job.com"
    country_id = random.choice(country_codes)
    dept = random.choice(departments)
    employee_id = employee_ids.pop()
    query.addBindValue(employee_id)
    query.addBindValue(f_name)
    query.addBindValue(l_name)
    query.addBindValue(email)
    query.addBindValue(dept)
    query.addBindValue(country_id)
    query.exec_()

# Create the second table, countries
country_query = QSqlQuery()
country_query.exec_("""CREATE TABLE countries (
            id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,
            country VARCHAR(20) NOT NULL)""")

country_query.prepare("INSERT INTO countries (country) VALUES (?)")
```

```
    # Add the values to the query to be inserted in countries
    for name in country_names:
        country_query.addBindValue(name)
        country_query.exec_()

    print("[INFO] Database successfully created.")

    sys.exit(0)

if __name__ == "__main__":
    CreateEmployeeData()
```

To see an example of what the data this program created looks like in a table view, refer back to Figure 10-6.

Explanation

This program does not create a GUI, so we only need to import the `QSqlDatabase` and `QSqlQuery` classes from `QtSql`. We will use QSqlDatabase to create the connection that allows access to a database; QSqlQuery can be used to perform SQL statements in PyQt.

We begin by creating a connection to the database in the `CreateEmployeeData` class. The `addDatabase()` function allows you to specify the SQL driver that you want to use. The examples in this chapter use SQLite 3 so we pass `QSQLITE` as the argument. Once the `database` object is created, we can set the other connection parameters, including which database we are going to use, the username, password, host name, and the connection port. For SQLite 3 we only need to specify the name of the database with `setDatabaseName()`. You can also create multiple connections to a database.

```
database = QSqlDatabase.addDatabase("QSQLITE")
database.setDatabaseName("files/accounts.db")
```

Note   A connection is referenced by its name, not by the name of the database. If you want to give your database a name, pass it as an argument after the driver in the `addDatabase()` method. If no name is specified, then that connection becomes the default connection.

If the `accounts.db` file does not already exist, then it will be created. Once the parameters are set, you must call `open()` to activate the connection to the database. A connection cannot be used until it is opened.

Now that the connections are established, we can begin querying our database. You typically might start with databases that already have data in them, but in this example, we are going to see how we can create a database using SQL commands. To query a database using PyQt, we first need to create an instance of `QSqlQuery`. Then, we call the `exec_()` method to execute the SQL statement in `query`. In the following lines, we want to delete the table `accounts`:

```
query = QSqlQuery()
query.exec_("DROP TABLE accounts")
```

Next, let's create a new `accounts` table using `exec_()` and `CREATE TABLE accounts`. Each table entry will have its own unique `id` by using `AUTOINCREMENT`. The accounts table will include information for an employee's ID, first name, last name, e-mail, department, and the country where they are located. We also create a `countries` table which holds the names of the employee's countries and is linked to the `accounts` table using the following line:

```
country_id VARCHAR(20) REFERENCES countries(id))
```

The `country_id` references the `countries` table's `id`. Figure 10-7 illustrates the connection between the two tables.
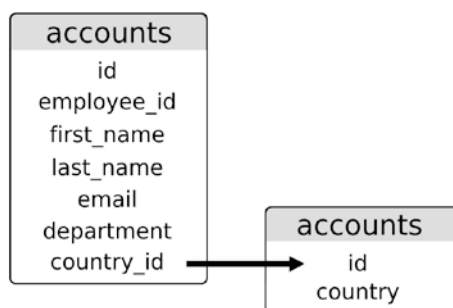
Figure 10-7. *Illustration of the relations between the accounts and countries tables*

The next thing to do is to insert records into our tables. We could continue to use exec_() to execute queries, but this would become tedious if we have a large database. To insert multiple records at the same time, we separate the query from the actual values being inserted using placeholders and the prepare() method. The placeholder will act as a temporary variable, allowing users to supply different data using the same SQL query. In the following code, the positional placeholders are the ?. PyQt supports two placeholder syntaxes – ODBC style which uses ? and the Oracle style which uses :field_name.

```
query.prepare("""INSERT INTO accounts (
              employee_id, first_name, last_name,
              email, department, country_id)
              VALUES (?, ?, ?, ?, ?, ?)""")
```

Each field, such as employee_id or first_name, is associated with one of the placeholders. Since we used AUTOINCREMENT for id, we do not have to include the field or a placeholder in the query.

The prepare() method gets the query ready for execution. If the query is prepared successfully, then values can be binded to the fields using the addBindValue() method.

Next, we create the values for the first_name, last_name, and other fields using Python lists and dictionaries. A for loop is then used where we bind the values to the placeholders. exec_() is called at the end of each iteration to insert the values into the accounts table. The countries table is prepared in a similar manner.

Once the tables are populated, we call sys.exit(0) to exit the program.

Example Queries Using QSqlQuery

The following code in Listing 10-4 is not necessary for the accounting manager GUI, but it does give a few more examples for understanding how to input, update, and delete records with SQL in a PyQt application.

Listing 10-4. Demonstrating how to insert, update, and delete records using SQL and PyQt

```
# query_examples.py
# Import necessary modules
import sys
from PyQt5.QtSql import QSqlDatabase, QSqlQuery
```

```python
class QueryExamples:

    def __init__(self):
        super().__init__()

        self.createConnection()
        self.exampleQueries()

    def createConnection(self):
        """
        Create connection to the database.
        """
        database = QSqlDatabase.addDatabase("QSQLITE")
        database.setDatabaseName("files/accounts.db")

        if not database.open():
            print("Unable to open data source file.")
            sys.exit(1) # Error code 1 - signifies error

    def exampleQueries(self):
        """
        Examples of working with the database.
        """
        # Executing a simple query
        query = QSqlQuery()
        query.exec_("SELECT first_name, last_name FROM accounts WHERE
        employee_id > 2000")

        # Navigating the result set
        while (query.next()):
            f_name = str(query.value(0))
            l_name = str(query.value(1))
            print(f_name, l_name)

        # Inserting a single new record into the database
        query.exec_("""INSERT INTO accounts (
                employee_id, first_name, last_name,
                email, department, country_id)
```

```
                VALUES (2134, 'Robert', 'Downey', 'downeyr@job.com',
                'Managerial', 1)""")

    # Update a record in the database
    query.exec_("UPDATE accounts SET department = 'R&D' WHERE employee_
    id = 2134")

    # Delete a record from the database
    query.exec_("DELETE FROM accounts WHERE employee_id <= 1500")

    sys.exit(0)

if __name__ == "__main__":
    QueryExamples()
```

This code will modify the database created in Listing 10-3. To view the changes, run
this code and then run the code in one of the following examples to see how the tables
have been manipulated.

Explanation

This example also has no GUI window. If you run this program after running the
program in Listing 10-3, you will notice how the queries here modify the database.

We start by creating a connection to the SQLite 3 driver and add the database created
in the previous program, accounts.db. Next, we complete the connection using open().

In exampleQueries(), let's take a look at how to use the QSqlQuery class and SQL
commands to query the database. We create a new QSqlQuery instance to search for the
first and last names of the employees whose employee IDs are greater than 2000.

```
query.exec_("SELECT first_name, last_name FROM accounts WHERE
employee_id > 2000")
```

With that query, we could use the values from first_name and last_name to update or
delete records. To cycle through the results of the query, we use next(). Other methods that
could be used to navigate the results include next(), previous(), first(), and last().

To insert a single record, we can use the INSERT SQL command. You could also add
multiple records into the database. Refer back to Listing 10-3 to see how. In this query, we
insert specific values for each field. To update records, use UPDATE. We update the department
value for the employee that was just inserted. Finally, to delete a record, use DELETE.

Working with QSqlTableModel

We are finally going to create a GUI for visualizing the database's contents. In this table, we are only going to visualize the accounts table to demonstrate the **QSqlTableModel** class, an interface that is useful for reading and writing database records when you only need to use a single table with no links to other tables. The following program will demonstrate how to use model/view programming to view the contents of a SQL database.

We could use QSqlQuery to do all of the database work, but combining the class with PyQt's model/view paradigm allows for us to design GUIs that make the data management process simpler.

Listing 10-5. Code to view SQL database using QSqlTableModel

```
# table_model.py
# Import necessary modules
import os, sys
from PyQt5.QtWidgets import (QApplication, QWidget, QTableView,
QVBoxLayout, QMessageBox, QHeaderView)
from PyQt5.QtSql import QSqlDatabase, QSqlTableModel

class TableDisplay(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(1000, 500)
        self.setWindowTitle('SQL Table Model')

        self.createConnection()
        self.createTable()

        self.show()
```

```python
def createConnection(self):
    """
    Set up the connection to the database.
    Check for the tables needed.
    """
    database = QSqlDatabase.addDatabase("QSQLITE")
    database.setDatabaseName("files/accounts.db")

    if not database.open():
        print("Unable to open data source file.")
        sys.exit(1) # Error code 1 - signifies error

    # Check if the tables we need exist in the database
    tables_needed = {'accounts'}
    tables_not_found = tables_needed - set(database.tables())
    if tables_not_found:
        QMessageBox.critical(None, 'Error',
            f'The following tables are missing from the database:
            {tables_not_found}')
        sys.exit(1) # Error code 1 – signifies error

def createTable(self):
    """
    Create the table using model/view architecture.
    """
    # Create the model
    model = QSqlTableModel()
    model.setTable('accounts')

    table_view = QTableView()
    table_view.setModel(model)
    table_view.horizontalHeader().setSectionResizeMode(QHeaderView.
    Stretch)
```
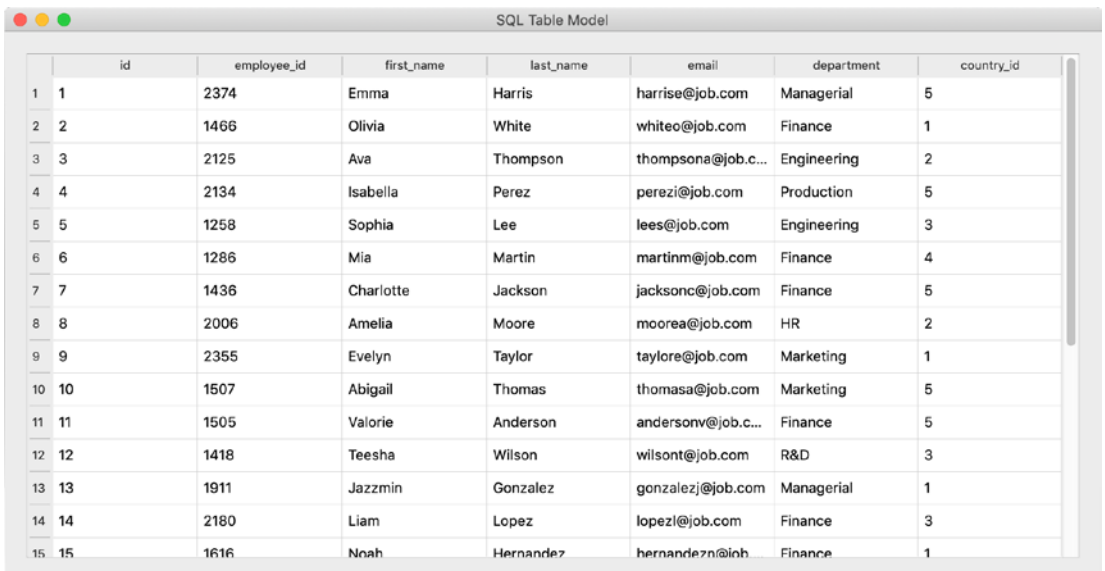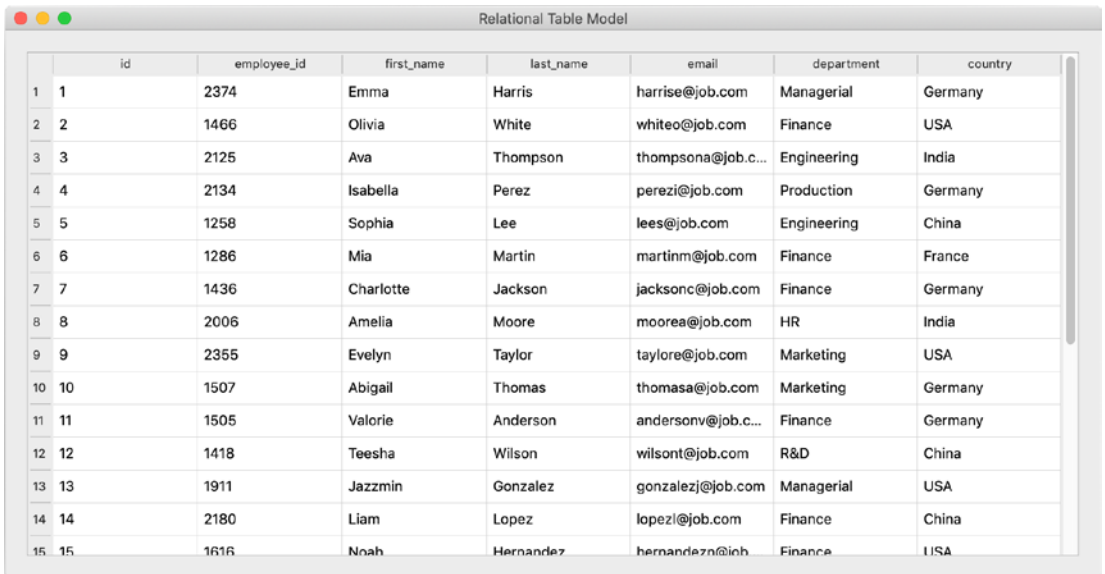
```
        # Populate the model with data
        model.select()
        # Main layout
        main_v_box = QVBoxLayout()
        main_v_box.addWidget(table_view)
        self.setLayout(main_v_box)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = TableDisplay()
    sys.exit(app.exec_())
```

Figure 10-8 displays the contents of the database in a table view. Notice how the header labels display the field names used when the database was created. We will see how to set header labels later. Also, the country_id column currently only displays numbers associated with the different names in the countries table. If you only want to display specific columns, the following code lets you select which ones you want to display:

```
model.setQuery(QSqlQuery("SELECT id, employee_id, first_name, last_name
FROM accounts"))
```


	id	employee_id	first_name	last_name	email	department	country_id
1	1	2374	Emma	Harris	harrise@job.com	Managerial	5
2	2	1466	Olivia	White	whiteo@job.com	Finance	1
3	3	2125	Ava	Thompson	thompsona@job.c...	Engineering	2
4	4	2134	Isabella	Perez	perezi@job.com	Production	5
5	5	1258	Sophia	Lee	lees@job.com	Engineering	3
6	6	1286	Mia	Martin	martinm@job.com	Finance	4
7	7	1436	Charlotte	Jackson	jacksonc@job.com	Finance	5
8	8	2006	Amelia	Moore	moorea@job.com	HR	2
9	9	2355	Evelyn	Taylor	taylore@job.com	Marketing	1
10	10	1507	Abigail	Thomas	thomasa@job.com	Marketing	5
11	11	1505	Valorie	Anderson	andersonv@job.c...	Finance	5
12	12	1418	Teesha	Wilson	wilsont@job.com	R&D	3
13	13	1911	Jazzmin	Gonzalez	gonzalezj@job.com	Managerial	1
14	14	2180	Liam	Lopez	lopezl@job.com	Finance	3
15	15	1616	Noah	Hernandez	hernandezn@job...	Finance	1

Figure 10-8. *The table created using QSqlTableModel*

305

Explanation

Get started by importing the PyQt classes, including `QSqlTableModel`. Next, create the `TableDisplay` class for displaying the contents of the database.

In the `createConnection()` method, we connect to the database and activate the connection with `open()`. This time, let's check to make sure that the tables we want to use are in the database. If they cannot be found, then a dialog box will be displayed to inform the user and the program will close.

The instances of the `QSqlTableModel` and the `QTableView` are created in the `createTable()` method. For the `model`, we need to set the database table we want to use with `setTable()`.

```
model.setTable('accounts')
```

Next, set the model for `table_view` using `setModel()`. To make the table stretch to fit into the view horizontally, we use the following line:

```
table_view.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)
```

This line also handles stretching the table when the window resizes.

Finally, populate the model with data using `select()`. If you have made changes to the table but have not submitted them, then `select()` will cause the edited items to return back to their previous states.

Working with QSqlRelationalTableModel

Next we are going to see how to use PyQt's QSqlRelationalTableModel for working with relational databases. The **QSqlRelationalTableModel** class provides a model for viewing and editing data in a SQL table, with support for using foreign keys. A foreign key is a SQL constraint used to link tables together. The application in Listing 10-6 builds upon the previous example in Listing 10-5.

Listing 10-6. Code to view SQL database using QSqlRelationalTableModel

```python
# relational_model.py
# Import necessary modules
import os, sys
from PyQt5.QtWidgets import (QApplication, QWidget, QTableView,
QVBoxLayout, QMessageBox, QHeaderView)
```

```python
from PyQt5.QtSql import (QSqlDatabase, QSqlRelationalTableModel,
QSqlRelation)

class TableDisplay(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(1000, 500)
        self.setWindowTitle('Relational Table Model')

        self.createConnection()
        self.createTable()

        self.show()
    def createConnection(self):
        """
        Set up the connection to the database.
        Check for the tables needed.
        """
        database = QSqlDatabase.addDatabase("QSQLITE")
        database.setDatabaseName("files/accounts.db")

        if not database.open():
            print("Unable to open data source file.")
            sys.exit(1) # Error code 1 - signifies error

        # Check if the tables we need exist in the database
        tables_needed = {'accounts', 'countries'}
        tables_not_found = tables_needed - set(database.tables())
        if tables_not_found:
            QMessageBox.critical(None, 'Error',
```

```python
                f'The following tables are missing from the database:
                {tables_not_found}')
            sys.exit(1) # Error code 1 - signifies error

    def createTable(self):
        """
        Create the table using model/view architecture.
        """
        # Create the model
        model = QSqlRelationalTableModel()
        model.setTable('accounts')
        # Set up relationship for foreign keys
        model.setRelation(model.fieldIndex('country_id'),
        QSqlRelation('countries', 'id', 'country'))

        table_view = QTableView()
        table_view.setModel(model)
        table_view.horizontalHeader().setSectionResizeMode(QHeaderView.
        Stretch)

        # Populate the model with data
        model.select()

        # Main layout
        main_v_box = QVBoxLayout()
        main_v_box.addWidget(table_view)
        self.setLayout(main_v_box)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = TableDisplay()
    sys.exit(app.exec_())
```

If you compare Figure 10-9 to Figure 10-8, you will notice that data in the last column has been updated to display the names of the countries and that the header has been changed to country.

Figure 10-9. *The table created using QSqlRelationalTableModel*

Explanation

This time we need to import `QSqlRelationalModel` since we are working with relational databases and foreign keys. Also, `QSqlRelation` stores the information about SQL foreign keys.

We connect to the database like before, except this time we are checking for both tables, `accounts` and `countries`. Next we create instances of the `QSqlRelationalModel` and `QTableView` classes. The `setTable()` method is used to cause the `model` to fetch the `accounts` table's information.

The `country_id` field in `accounts` is mapped to `countries`' field ID. Using `setRelation()`, we can cause `table_view` to present the `countries`' country field to the user. The following code shows how to do this, and the results can be seen in Figure 10-9:

```
model.setRelation(model.fieldIndex('country_id'), QSqlRelation('countries',
'id', 'country'))
```

The rest of the program is the same as Listing 10-5.

309

Account Management GUI Solution

The account management GUI uses the QSqlRelationalModel for managing the accounts and countries tables. We use the concepts we learned in the previous sections and design a GUI with features for managing the database directly rather than programmatically.

The account management GUI lets a user add, delete, and sort the contents of the table. Rows added or deleted will also update the database. This example also briefly shows how to create a delegate for editing data. The code for the account management GUI can be found in Listing 10-7.

Listing 10-7. Code for the account management GUI

```python
# account_manager.py
# Import necessary modules
import sys, os
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
    QPushButton, QComboBox, QTableView, QHeaderView,
    QHBoxLayout, QVBoxLayout, QSizePolicy, QMessageBox)
from PyQt5.QtSql import (QSqlDatabase, QSqlQuery,
    QSqlRelationalTableModel, QSqlRelation,
    QSqlRelationalDelegate)
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QIcon

class AccountManager(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(1000, 600)
        self.setWindowTitle('10.1 - Account Management GUI')
```

```python
    self.createConnection()
    self.createTable()
    self.setupWidgets()

    self.show()

def createConnection(self):
    database = QSqlDatabase.addDatabase("QSQLITE") # SQLite version 3
    database.setDatabaseName("files/accounts.db")

    if not database.open():
        print("Unable to open data source file.")
        sys.exit(1) # Error code 1 - signifies error

    # Check if the tables we need exist in the database
    tables_needed = {'accounts', 'countries'}
    tables_not_found = tables_needed - set(database.tables())
    if tables_not_found:
        QMessageBox.critical(None, 'Error',
            f'The following tables are missing from the database:
            {tables_not_found}')
        sys.exit(1) # Error code 1 - signifies error

def createTable(self):
    """
    Set up the model, headers and populate the model.
    """
    self.model = QSqlRelationalTableModel()
    self.model.setTable('accounts')
    self.model.setRelation(self.model.fieldIndex('country_id'),
    QSqlRelation('countries', 'id', 'country'))

    self.model.setHeaderData(self.model.fieldIndex('id'),
    Qt.Horizontal, "ID")
    self.model.setHeaderData(self.model.fieldIndex('employee_id'),
    Qt.Horizontal, "Employee ID")
    self.model.setHeaderData(self.model.fieldIndex('first_name'),
    Qt.Horizontal, "First")
```

311

```python
        self.model.setHeaderData(self.model.fieldIndex('last_name'),
        Qt.Horizontal, "Last")
        self.model.setHeaderData(self.model.fieldIndex('email'),
        Qt.Horizontal, "E-mail")
        self.model.setHeaderData(self.model.fieldIndex('department'),
        Qt.Horizontal, "Dept.")
        self.model.setHeaderData(self.model.fieldIndex('country_id'),
        Qt.Horizontal, "Country")

        # Populate the model with data
        self.model.select()

    def setupWidgets(self):
        """
        Create instances of widgets, the table view and set layouts.
        """
        icons_path = "icons"

        title = QLabel("Account Management System")
        title.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
        title.setStyleSheet("font: bold 24px")

        add_record_button = QPushButton("Add Employee")
        add_record_button.setIcon(QIcon(os.path.join(icons_path, "add_user.
        png")))
        add_record_button.setStyleSheet("padding: 10px")
        add_record_button.clicked.connect(self.addRecord)

        del_record_button = QPushButton("Delete")
        del_record_button.setIcon(QIcon(os.path.join(icons_path, "trash_
        can.png")))
        del_record_button.setStyleSheet("padding: 10px")
        del_record_button.clicked.connect(self.deleteRecord)

        # Set up sorting combo box
        sorting_options = ["Sort by ID", "Sort by Employee ID", "Sort by
        First Name", "Sort by Last Name", "Sort by Department", "Sort by
        Country"]
```

```python
        sort_name_cb = QComboBox()
        sort_name_cb.addItems(sorting_options)
        sort_name_cb.currentTextChanged.connect(self.setSortingOrder)

        buttons_h_box = QHBoxLayout()
        buttons_h_box.addWidget(add_record_button)
        buttons_h_box.addWidget(del_record_button)
        buttons_h_box.addStretch()
        buttons_h_box.addWidget(sort_name_cb)

        # Widget to contain editing buttons
        edit_buttons = QWidget()
        edit_buttons.setLayout(buttons_h_box)

        # Create table view and set model
        self.table_view = QTableView()
        self.table_view.setModel(self.model)
        self.table_view.horizontalHeader().setSectionResizeMode(QHeaderVie
        w.Stretch)
        self.table_view.verticalHeader().setSectionResizeMode(QHeaderView.
        Stretch)
        self.table_view.setSelectionMode(QTableView.SingleSelection)
        self.table_view.setSelectionBehavior(QTableView.SelectRows)
        # Instantiate the delegate
        delegate = QSqlRelationalDelegate(self.table_view)
        self.table_view.setItemDelegate(delegate)

        # Main layout
        main_v_box = QVBoxLayout()
        main_v_box.addWidget(title, Qt.AlignLeft)
        main_v_box.addWidget(edit_buttons)
        main_v_box.addWidget(self.table_view)
        self.setLayout(main_v_box)

    def addRecord(self):
        """
        Add a new record to the last row of the table.
        """
```

```python
        last_row = self.model.rowCount()
        self.model.insertRow(last_row)

        id = 0
        query = QSqlQuery()
        query.exec_("SELECT MAX (id) FROM accounts")
        if query.next():
            id = int(query.value(0))

    def deleteRecord(self):
        """
        Delete an entire row from the table.
        """
        current_item = self.table_view.selectedIndexes()
        for index in current_item:
            self.model.removeRow(index.row())
        self.model.select()

    def setSortingOrder(self, text):
        """
        Sort the rows in table.
        """
        if text == "Sort by ID":
            self.model.setSort(self.model.fieldIndex('id'),
            Qt.AscendingOrder)
        elif text == "Sort by Employee ID":
            self.model.setSort(self.model.fieldIndex('employee_id'),
            Qt.AscendingOrder)
        elif text == "Sort by First Name":
            self.model.setSort(self.model.fieldIndex('first_name'),
            Qt.AscendingOrder)
        elif text == "Sort by Last Name":
            self.model.setSort(self.model.fieldIndex('last_name'),
            Qt.AscendingOrder)
        elif text == "Sort by Department":
            self.model.setSort(self.model.fieldIndex('department'),
            Qt.AscendingOrder)
```

```
        elif text == "Sort by Country":
            self.model.setSort(self.model.fieldIndex('country'),
            Qt.AscendingOrder)

        self.model.select()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = AccountManager()
    sys.exit(app.exec_())
```

Your GUI should look similar to the one displayed in Figure 10-6.

Explanation

After importing all of the PyQt classes we need and setting up the AccountManager class, next we need to connect to the accounts database just like we have previously done. The createTable() method instantiates and sets up the model, creating the foreign key between the two tables. The setHeaderData() method applies labels to each of the columns by using fieldIndex() to locate the index of the given field name. An example is given as follows:

```
self.model.setHeaderData(self.model.fieldIndex('id'), Qt.Horizontal, "ID")
```

The QTableView object, table_view, is created in the setupWidgets() method, along with the GUI's labels, push buttons, and combo box. For table_view, we set the model and a few parameters. The table's vertical and horizontal headers will stretch to fit the window. QAbstractItemView.SingleSelection only allows the user to select one item at a time. QAbstractItemView.SelectRows only allows rows to be selected in the table.

The two push buttons, add_record_button and del_record_button, emit signals that add and delete rows in the table. For addRecord(), we check how many rows are in the table with rowCount() and use insertRow() to insert an empty row at the end of table view. We query the database to find out the largest id value. If a user does not enter a value for id into the row, then the new record's id is equal to the highest id value plus one. For deleteRecord(), we get the currently selected row's index and delete the row with removeRow(). Then we update the model using select().

315

For the QComboBox, when the selection has changed, the widget emits a currentTextChanged() signal. We use the text to determine how to set the view's order for displaying records.

In the model/view architecture, the delegate provides the default tools for painting item data in the view and for providing editor widgets for item models. The appearance and editor widgets of the item delegate can be customized. For the account management GUI, the delegate used is the **QSqlRelationalDelegate**. This class provides a combo box for editing data in fields that are foreign keys for other tables.

```
delegate = QSqlRelationalDelegate(self.table_view)
self.table_view.setItemDelegate(delegate)
```

An example of the combo box used by the delegate can be seen in the bottom-right corner of Figure 10-6. The widget appears whenever the user needs to select a country from the countries table that will be displayed in the view.

Summary

PyQt provides convenience classes for lists, tables, and trees. QListWidget, QTableWidget, and QTreeWidget are useful when you need to view data for general situations. While they are practical for creating quick interfaces for editing data, if you need to have more than one widget for displaying a dataset in an application, you must also create a process for keeping the datasets and the widgets in agreement. A better option is to use PyQt's model/view architecture.

With the model/view paradigm, you are able to have multiple views in a single application that work in unison to view and update the database. You also have more control over the look of the editing widgets and the items in the view with the delegate.

There are different formats available for storing and managing data. One example is the CSV format which is convenient for reading, parsing, and storing smaller datasets. However, for large databases that contain multiple tables with relational characteristics, a relational database management system that uses SQL is a more preferable option for managing the data. SQL allows users to select desired information that might be shared between tables, as well as insert, update, and delete existing records easily.

The model/view architecture is very useful for working with SQL databases, providing the tools necessary for connecting to a database and viewing its content. PyQt provides three models for working with SQL databases. For an editable data model

without foreign key support, use QSqlTableModel. If you have tables with relational properties, use QSqlRelationalTableModel. Finally, the **QSqlQueryModel** is beneficial when you only need to read the results of a query without editing them.

Over the course of this book, we took a look at a few applications that could have benefited greatly by being able to connect to databases using SQL. The login GUI in Chapter 3 could connect to a database to retrieve usernames and passwords. The to-do list GUI in Chapter 4 could be completely redesigned to include a QCalendarWidget (covered in Chapter 12) that keeps track of events by using a database. There is also the pizza ordering GUI from Chapter 6. You could implement a database for storing customers' information, using a relational database for adding new customers, updating existing ones, and preventing data from being duplicated.

In Chapter 11, we will take a brief look at multithreading in PyQt.

Managing Threads

We have all experienced that moment when running some process such as copying files between directories or launching a new instance of an application causes a program to lag for just a moment and, in some cases, to freeze completely. We are then forced to either wait for the current task to complete or Ctrl+Alt+Delete our way to freedom. When you are creating GUIs, you should be aware of how to handle, or more preferably have foresight about avoiding, these situations.

The motivation behind this chapter is twofold – to help you design more robust GUI applications and to inform you of how you might be able to handle situations where your applications need to run long processes. Any action that causes event processing in an application to come to a standstill is bad for a user's experience.

This chapter takes a look at

- How to implement threading with `QThread`

- A few other techniques for handling time-consuming processes

- The `QProgressBar` widget for giving visual feedback about a task's progression

Introduction to Threading

A computer's performance can be measured by the accuracy, efficiency, and speed at which it can execute program instructions. Modern computers can take advantage of their multicore processors to run those instructions in parallel, thereby increasing the performance of computer applications that have been written to take advantage of the multicore architecture.

The idea of performing tasks in a synchronous manner, that is, where only one task is processed at a time until completion before moving on to the next task, can be inefficient, especially for larger operations. What we need is a way to perform operations concurrently. That is where threads and processes come into play.

Threads and processes are not the same thing. Without going too much into the technical jargon, let's try and understand the differences between the two. A **process** is an instance of an application that requires memory and computer resources to run. Opening up the word processor on your computer to write an essay is one process. While writing your essay, you also need to search on the Internet for information. You now have two separate processes running on your computer independently and in parallel. What happens in one process is not influencing the other. Of course, you have multiple tabs open in the web browser, and each tab is loading and updating information; those tabs are working side by side with the web browser. This is where a thread becomes important.

A **thread** is essential to the concurrency within an individual process. When a process begins, it only has one thread, and multiple threads can be started within a single process. These threads, just like the processes, are managed by the CPU. **Multithreading** occurs when the CPU can handle multiple threads of execution concurrently within one process. These threads are independent but also share the process's resources. Using multithreading allows for applications to be more responsive to user's inputs while other operations are occurring in the background, and to better utilize a system's resources.

On a system with only a single CPU, true parallelism is actually unachievable. In these instances, the CPU is shared among the processes or threads. To switch between threads, context switches are used to interrupt the current thread, save its state, and then restore the next thread's state. This gives the user a false appearance of parallelism.

To achieve true parallelism and create a truly concurrent system, a multicore processor would allow threads in a multithreaded application to be assigned to different processors.

Threading in PyQt

Applications based on Qt are event based. When the event loop is started using exec_(), a thread is created. This thread is referred to as the main thread of the GUI. Any events that take place in the main thread, including the GUI itself, run synchronously within the main event loop. To take advantage of threading, we need to create a secondary thread to offload processing operations from the main thread.

PyQt makes communicating between the main thread and secondary threads, also referred to as worker threads, simple with signals and slots. This can be useful for relaying feedback, allowing the user to interrupt a process, and for informing the main thread that a process has finished. Since threads utilize the same address space, they can share data very easily.

However, if multiple threads try to access shared data or resources concurrently, this can cause crashes or memory corruption. Deadlock is another issue that can occur if two threads are blocked because they are waiting for resources. PyQt provides a few classes, for example, QMutex, QReadWriteLock, and QSemaphore, for avoiding these kinds of problems.

Note   Python also has a number of modules for handling threading and processing tasks, including `_thread`, `threading`, `asyncio`, and `multiprocessing`. While you can also use this modules, PyQt's `QThread` and other classes allow you to emit signals between the main and worker threads.

Methods for Processing Long Events in PyQt

While this chapter focuses on using QThread, it is also a good idea to keep in mind that there are also other ways that you might want to try before attempting to use threading in your GUI. Implementing threading can lead to problems with concurrency and identifying errors. Combined with signals and slots, PyQt provides a few different ways to handle time-consuming operations.

Choosing which method is best for your application comes down to considering your situation. The main methods, including threading, for handling these kinds of events are listed as follows:

1. If there is a process in your application that is causing it to freeze, check to see if that process can be broken down into smaller steps and perform them sequentially. Manually handle the processing of long operations, and explicitly call `QApplication.processEvents()` to process pending events. This works best if your operations can be processed using a single thread.

2. With `QTimer` and signals and slots, you can schedule operations to be performed at certain intervals in the future.

3. Use QThread to create a worker thread that will perform long
    operations in a separate thread. Derive a class from QThread,
    reimplement run(), and use PyQt's signal and slot mechanism
    to communicate with the main thread. This method can help to
    avoid blocking the main event loop.

4. The QThreadPool and QRunnable classes can be used to divide
    the work across the CPUs on your computer. Create a subclass of
    QRunnable and reimplement the run() function; an instance of
    QRunnable can then be passed to threads that are managed by
    QThreadPool. QThreadPool handles the queuing and execution of
    QRunnable instances for you.

There are even other options that may depend upon your application's
requirements. Keep in mind that, while using threads could benefit your application,
they could also slow it down or cause errors if used incorrectly.

Project 11.1 – File Renaming GUI

This chapter's project, shown in Figure 11-1, actually stems from my own experiences.
Creating datasets for training neural networks often entails writing Python scripts
for labeling thousands of images and data files. Those scripts are generally written to
include some kind of visual feedback to the user about how the process is going in the
command line.

For this project, we are going to create a GUI that will allow us to select a local directory
and edit the names of files with the specified extension. The interface includes QTextEdit and
QProgressBar widgets as two different means of feedback about the file labeling process. This
application also takes advantage of the QThread class so that users are still able to interact
with the interface while the operations are being performed in the background.
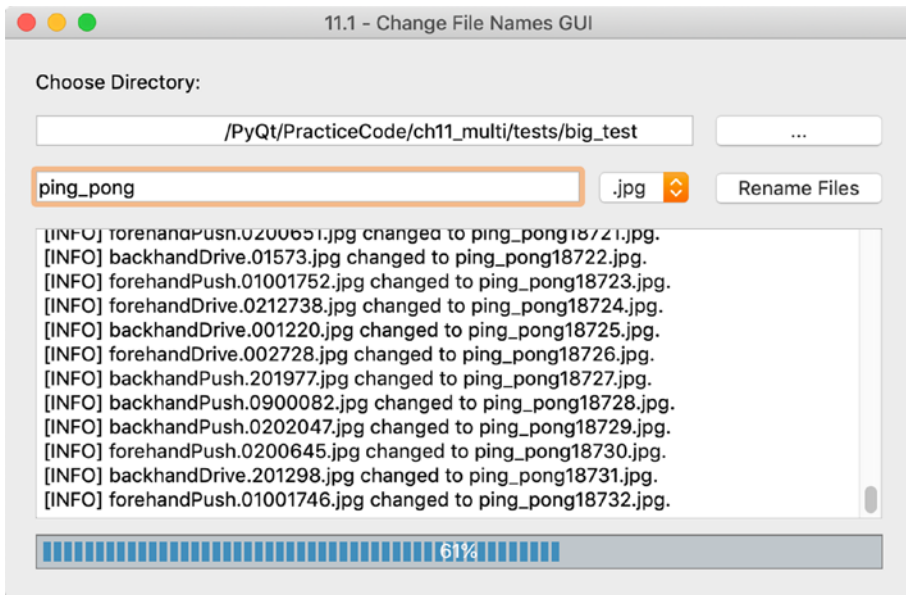
Figure 11-1. *The interface for renaming files in a selected directory*

The QProgressBar Widget

The QProgressBar widget visually relays the progress of an extended operation back to the user. This feedback can also be used as reassurance that a process, such as a download, installation, or file transfer, is still running. Some of the settings that can be controlled include the widget's orientation and range.

Refer to the project in this chapter for setting up the progress bar.

File Renaming GUI Solution

The GUI window contains various buttons and editor widgets that allow the user to manage file renaming. The user can select a directory using a QFileDialog. They can also enter the new file name in the QLineEdit widget. Using the combo box, they can select the file extension for the files they want to change.

The application uses threading to update the progress bar and display information about the files being changed in the text edit and performs the actual renaming operation. This is all done using signals and slots. The code for the file renaming application can be found in Listing 11-1.

Listing 11-1. Code for the GUI that renames files in a directory using threading

```python
# file_rename_threading.py
import os, sys, time
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QProgressBar,
QLineEdit, QPushButton, QTextEdit, QComboBox, QFileDialog, QGridLayout)
from PyQt5.QtCore import pyqtSignal, QThread

style_sheet = """
    QProgressBar{
        background-color: #C0C6CA;
        color: #FFFFFF;
        border: 1px solid grey;
        padding: 3px;
        height: 15px;
        text-align: center;
    }

    QProgressBar::chunk{
        background: #538DB8;
        width: 5px;
        margin: 0.5px
    }
"""

# Create worker thread for running tasks like updating the progress bar,
renaming photos,
# displaying information in the text edit widget
class Worker(QThread):
    updateValueSignal = pyqtSignal(int)
    updateTextEditSignal = pyqtSignal(str, str)

    def __init__(self, dir, ext, prefix):
        super().__init__()
        self.dir = dir
        self.ext = ext
        self.prefix = prefix
```

```python
    def run(self):
        """
        The thread begins running from here. run() is only called after
        start().
        """
        for (i, file) in enumerate(os.listdir(self.dir)):
            _, file_ext = os.path.splitext(file)
            if file_ext == self.ext:
                new_file_name = self.prefix + str(i) + self.ext
                src_path = os.path.join(self.dir, file)
                dst_path = os.path.join(self.dir, new_file_name)

                # os.rename(src, dst): src is original address of file to
                #   be renamed
                # and dst is destination location with new name.
                os.rename(src_path, dst_path)
                #time.sleep(0.2) # Uncomment if process is too fast and
                want to see the updates.

                self.updateValueSignal.emit(i + 1)
                self.updateTextEditSignal.emit(file, new_file_name)
            else:
                pass
        self.updateValueSignal.emit(0) # Reset the value of the progress bar

class RenameFilesGUI(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(600, 250)
        self.setWindowTitle('11.1 - Change File Names GUI')
```

```python
        self.directory = ""
        self.cb_value = ""

        self.setupWidgets()

        self.show()

    def setupWidgets(self):
        """
        Set up the widgets and layouts for interface.
        """
        dir_label = QLabel("Choose Directory:")
        self.dir_line_edit = QLineEdit()
        dir_button = QPushButton('...')
        dir_button.setToolTip("Select file directory.")
        dir_button.clicked.connect(self.setDirectory)

        self.change_name_edit = QLineEdit()
        self.change_name_edit.setToolTip("Files will be appended with
        numerical values. For example: filename<b>01</b>.jpg")
        self.change_name_edit.setPlaceholderText("Change file names to...")

        rename_button= QPushButton("Rename Files")
        rename_button.setToolTip("Begin renaming files in directory.")
        rename_button.clicked.connect(self.renameFiles)

        file_exts = [".jpg", ".jpeg", ".png", ".gif", ".txt"]

        # Create combo box for selecting file extensions.
        ext_cb = QComboBox()
        self.cb_value = file_exts[0]
        ext_cb.setToolTip("Only files with this extension will be changed.")
        ext_cb.addItems(file_exts)
        ext_cb.currentTextChanged.connect(self.updateCbValue)

        # Text edit is for displaying the file names as they are updated.
        self.display_files_edit = QTextEdit()
        self.display_files_edit.setReadOnly(True)

        self.progress_bar = QProgressBar()
        self.progress_bar.setValue(0)
```

```python
    # Set layout and widgets.
    grid = QGridLayout()
    grid.addWidget(dir_label, 0, 0)
    grid.addWidget(self.dir_line_edit, 1, 0, 1, 2)
    grid.addWidget(dir_button, 1, 2)
    grid.addWidget(self.change_name_edit, 2, 0)
    grid.addWidget(ext_cb, 2, 1)
    grid.addWidget(rename_button, 2, 2)
    grid.addWidget(self.display_files_edit, 3, 0, 1, 3)
    grid.addWidget(self.progress_bar, 4, 0, 1, 3)

    self.setLayout(grid)

def setDirectory(self):
    """
    Choose the directory.
    """
    file_dialog = QFileDialog(self)
    file_dialog.setFileMode(QFileDialog.Directory)
    self.directory = file_dialog.getExistingDirectory(self, "Open
    Directory", "", QFileDialog.ShowDirsOnly)

    if self.directory:
        self.dir_line_edit.setText(self.directory)

        # Set the max value of progress bar equal to max number of
          files in the directory.
        num_of_files = len([name for name in os.listdir(self.
        directory)])
        self.progress_bar.setRange(0, num_of_files)

def updateCbValue(self, text):
    """
    Change the combo box value. Values represent the different file
    extensions.
    """
    self.cb_value = text
```

```
    def renameFiles(self):
        """
        Create instance of worker thread to handle the file renaming
        process.
        """
        prefix_text = self.change_name_edit.text()

        if self.directory != "" and prefix_text != "":
            self.worker = Worker(self.directory, self.cb_value, prefix_
            text)
            self.worker.updateValueSignal.connect(self.updateProgressBar)
            self.worker.updateTextEditSignal.connect(self.updateTextEdit)
            self.worker.start()
        else:
            pass

    def updateProgressBar(self, value):
        self.progress_bar.setValue(value)

    def updateTextEdit(self, old_text, new_text):
        self.display_files_edit.append("[INFO] {} changed to
        {}.".format(old_text, new_text))

if __name__ == "__main__":
    app = QApplication(sys.argv)
    app.setStyleSheet(style_sheet)
    window = RenameFilesGUI()
    sys.exit(app.exec_())
```

The application's GUI can be seen in Figure 11-1.

Explanation

We start with importing Python and PyQt classes. The style sheet is used to modify the appearance of the QProgressBar.

Let's start by looking at the RenameFileGUI class. Here we set up the window and other widgets, including push buttons for selecting the directory and starting the process

for renaming files, line edit widgets, and the text edit and the progress bar widgets for relaying feedback.

The user can select a directory using QFileDialog. Once a directory is chosen, the user can enter the new file names into change_name_edit and select the file extension for the types of files to change in the combo box.

Renaming the files could take place in the main thread. This wouldn't be a problem for a few files. However, if the user wants to work with a large number of files, this would cause the GUI to be locked until the operations are finished. Therefore, the process for renaming the files, along with updating the progress bar and the text edit widgets, is performed in the worker thread.

For this project, we subclass QThread. An instance of the QThread class manages only one thread. Two custom signals are created for updating the progress bar and text edit widgets.

```
updateValueSignal = pyqtSignal(int)
updateTextEditSignal = pyqtSignal(str, str)
```

The reimplemented QThread method run() begins executing the thread. The time-consuming operations – traversing the directory, renaming files, and emitting the signals for updating the QProgressBar and QTextEdit – are performed in run(). However, this method is not called directly. The QThread method start() is used to communicate with the worker thread and begin executing the thread by calling run(). The start() method is called in renameFiles().

Summary

Preventing GUIs from becoming frozen while processing long operations is important for a user's experience. There are a few options for effectively handling blocking in your application, including using timers and threads. PyQt makes using threading seem relatively simple with QThread and the signal and slot mechanism. However, you must be careful when using QThread to ensure that threads protect access to their own data. While not displayed in this chapter's short project, QThread also has methods, such as started(), finished(), wait(), and quit(), for managing threads.

In Chapter 12, we will take a look at an array of projects that utilize different PyQt classes.

CHAPTER 12

Extra Projects

This book has tried to take a practical approach to creating GUIs. As you use PyQt5 and Python more and more, you will find yourself learning about other modules and classes that you will need in your applications. Each chapter set up an idea and worked hard to break those projects down into their fundamental parts so that you could learn new ideas along the way.

PyQt5 has quite a few modules for a variety of purposes, and the chapters in this book only scratched the surface of the many possibilities for designing GUIs.

In Chapter 12, we will take a look at a few extra examples to give you ideas for other projects or to help you in creating new types of user interfaces. These projects will not go into great lengths of detail, but rather focus on explaining the key points of each new program and leave it up to you to research the details that you are unsure about, either by finding the answers in a different chapter or by searching online for help.

The projects in this chapter will take a look at the following concepts:

- Displaying directories and files using the `QFileSystemModel` class

- Working with multiple-document interface (MDI) applications and the `QCamera` class

- Creating a simple clock GUI with `QDate` and `QTime`

- Exploring the `QCalendarWidget` class

- Building Hangman with `QPainter` and other PyQt classes

- Building the framework for a web browser using the `QtWebEngineWidgets` module

Project 12.1 – Directory Viewer GUI

For every operating system, there needs to be some method for a user to access the data and files located in it. These files are stored in a hierarchical file system, displaying drives, directories, and files in groups so that you only view the files that you are interested in seeing.

Whether you use a command-line interface or a graphical user interface, there needs to be some way to create, remove, and rename files and directories. However, if you are already interacting with one interface, it may be more convenient to locate files or directories that you need in your current application rather than opening new windows or other programs.

This project shows you how to set up an interface for viewing the files on your local system. There are two key classes that will be introduced in this project – **QFileSystemModel**, which grants you access to the file system on your computer, and **QTreeView**, which provides a visual representation of data using a treelike structure (Listing 12-1).

Listing 12-1. Code for directory viewer GUI

```
# display_directory.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QFileSystemModel,
QTreeView, QFrame, QAction, QFileDialog, QVBoxLayout)

class DisplayDirectory(QMainWindow):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(500, 400)
        self.setWindowTitle('12.1 - View Directory GUI')
```

```python
        self.createMenu()
        self.setupTree()

        self.show()

    def createMenu(self):
        """
        Set up the menu bar.
        """
        open_dir_act = QAction('Open Directory...', self)
        open_dir_act.triggered.connect(self.chooseDirectory)

        root_act = QAction("Return to Root", self)
        root_act.triggered.connect(self.returnToRootDirectory)

        # Create menubar
        menu_bar = self.menuBar()
        #menu_bar.setNativeMenuBar(False) # Uncomment for MacOS

        # Create file menu and add actions
        dir_menu = menu_bar.addMenu('Directories')
        dir_menu.addAction(open_dir_act)
        dir_menu.addAction(root_act)

    def setupTree(self):
        """
        Set up the QTreeView so that it displays the contents
        of the local filesystem.
        """
        self.model = QFileSystemModel()
        self.model.setRootPath('')

        self.tree = QTreeView()
        self.tree.setIndentation(10) # Indentation of items
        self.tree.setModel(self.model)
```

```python
        # Set up container and layout
        frame = QFrame()
        frame_v_box = QVBoxLayout()
        frame_v_box.addWidget(self.tree)
        frame.setLayout(frame_v_box)

        self.setCentralWidget(frame)

    def chooseDirectory(self):
        """
        Select a directory to display.
        """
        file_dialog = QFileDialog(self)
        file_dialog.setFileMode(QFileDialog.Directory)
        directory = file_dialog.getExistingDirectory(self, "Open
        Directory", "", QFileDialog.ShowDirsOnly)

        self.tree.setRootIndex(self.model.index(directory))

    def returnToRootDirectory(self):
        """
        Re-display the contents of the root directory.
        """
        self.tree.setRootIndex(self.model.index(''))

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = DisplayDirectory()
    sys.exit(app.exec_())
```

The directory viewer application can be seen in Figure 12-1.
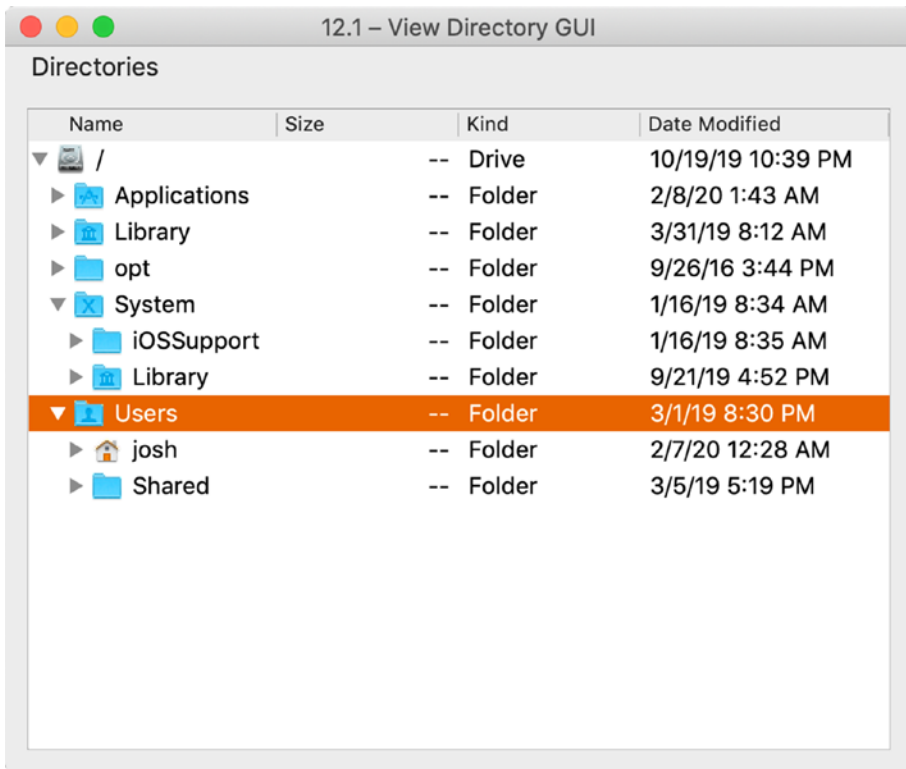
Figure 12-1. *Directory viewer displaying the local system's directories*

Explanation

Begin by importing the necessary modules for this GUI. For this project, we will need to use the model/view paradigm to view the data on your computer. For more information about model/view programming, refer to Chapter 10.

The QFileSystemModel class provides the model we need to access data on the local file system. While not included in this project, you could also use QFileSystemModel to rename or remove files and directories, create new directories, or use it with other display widgets as part of a browser.

The QTreeView class will be used to display the contents of the model in a hierarchical tree view.

For this GUI, we will create a Directories menu with actions that will either let the user view a specific directory or return back to the root directory. The menu system can be seen in Figure 12-2.

Create an instance of the QFileSystemModel class, model, and set the directory to the root path on your system.

```
self.model.setRootPath('') # Sets path to system's root path
```

Set the model for the tree object to show the contents of the file system using setModel(). To choose a different directory, the user can select Open Directory... from the menu and a file dialog will appear. A new directory can then be selected and set as the new root path to be displayed in the tree object.

```
self.tree.setRootIndex(self.model.index(directory))
```


Figure 12-2.   *The menu for the directory viewer GUI*

Project 12.2 – Camera GUI

When creating GUIs, there are a number of ways to tackle the issue of interfaces with multiple windows. You could use stacked or tabbed widgets, but these methods only allow for one window to be displayed at a time. Another option is to use dock widgets and allow windows to be floatable or used as secondary windows.

For this project, you will see how to set up a multiple-windowed GUI using the **QMdiArea** class. QMdiArea provides the area for displaying MDI windows. Multiple-document interface (MDI) is a type of interface that allows users to work with multiple windows at the same time. MDI applications require less memory resources and make the process of laying out subwindows much simpler.

Let's take a look at how to use the **QCamera** class to create an MDI application (Listing 12-2).

Listing 12-2. Example code to show how to create MDI applications

```python
# camera.py
# Import necessary modules
import os, sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QListWidget,
QListWidgetItem, QLabel, QGroupBox, QPushButton, QVBoxLayout, QMdiArea,
QMdiSubWindow,)
from PyQt5.QtMultimedia import QCamera, QCameraInfo, QCameraImageCapture
from PyQt5.QtMultimediaWidgets import QCameraViewfinder
from PyQt5.QtCore import Qt

class Camera(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen
        """
        self.setGeometry(100, 100, 600, 400)
        self.setWindowTitle('12.2 – Camera GUI')

        self.setupWindows()

        self.show()

    def setupWindows(self):
        """
        Set up QMdiArea parent and subwindows.
        Add available cameras on local system as items to
        list widget.
        """
        # Create images directory if it does not already exist
        path = 'images'
        if not os.path.exists(path):
            os.makedirs(path)
```

```python
# Set up list widget that will display identified
# cameras on your computer.
picture_label = QLabel("Press 'Spacebar' to take pictures.")
camera_label = QLabel("Available Cameras")
self.camera_list_widget = QListWidget()
self.camera_list_widget.setAlternatingRowColors(True)

# Add availableCameras to a list to be displayed in
# list widget. Use QCameraInfo() to list available cameras.
self.cameras = list(QCameraInfo().availableCameras())
for camera in self.cameras:
    self.list_item = QListWidgetItem()
    self.list_item.setText(camera.deviceName())
    self.camera_list_widget.addItem(self.list_item)

# Create button that will allow user to select camera
choose_cam_button = QPushButton("Select Camera")
choose_cam_button.clicked.connect(self.selectCamera)

# Create child widgets and layout for camera controls subwindow
controls_gbox = QGroupBox()
controls_gbox.setTitle("Camera Controls")

v_box = QVBoxLayout()
v_box.addWidget(picture_label, alignment=Qt.AlignCenter)
v_box.addWidget(camera_label)
v_box.addWidget(self.camera_list_widget)
v_box.addWidget(choose_cam_button)

controls_gbox.setLayout(v_box)

controls_sub_window = QMdiSubWindow()
controls_sub_window.setWidget(controls_gbox)
controls_sub_window.setAttribute(Qt.WA_DeleteOnClose)

# Create viewfinder subwindow
self.view_finder_window = QMdiSubWindow()
self.view_finder_window.setWindowTitle("Camera View")
self.view_finder_window.setAttribute(Qt.WA_DeleteOnClose)
```

```python
    # Create QMdiArea widget to manage subwindows
    mdi_area = QMdiArea()
    mdi_area.tileSubWindows()
    mdi_area.addSubWindow(self.view_finder_window)
    mdi_area.addSubWindow(controls_sub_window)

    # Set mdi_area widget as the central widget of main window
    self.setCentralWidget(mdi_area)

def setupCamera(self, cam_name):
    """
    Create and setup camera functions.
    """
    for camera in self.cameras:
        # Select camera by matching cam_name to one of the
        # devices in the cameras list.
        if camera.deviceName() == cam_name:
            self.cam = QCamera(camera) # Construct QCamera device

            # Create camera viewfinder widget and add it to the
            #   view_finder_window.
            self.view_finder = QCameraViewfinder()
            self.view_finder_window.setWidget(self.view_finder)
            self.view_finder.show()

            # Sets the viewfinder to display video
            self.cam.setViewfinder(self.view_finder)

            # QCameraImageCapture() is used for taking
            # images or recordings.
            self.image_capture = QCameraImageCapture(self.cam)

            # Configure the camera to capture still images.
            self.cam.setCaptureMode(QCamera.CaptureStillImage)
            self.cam.start() # Slot to start the camera
        else:
            pass
```

```python
    def selectCamera(self):
        """
        Slot for selecting one of the available cameras displayed in list
        widget.
        """
        try:
            if self.list_item.isSelected():
                camera_name = self.list_item.text()
                self.setupCamera(camera_name)
            else:
                print("No camera selected.")
                pass
        except:
            print("No cameras detected.")

    def keyPressEvent(self, event):
        """
        Handle the key press event so that the camera takes images.
        """
        if event.key() == Qt.Key_Space:
            try:
                self.cam.searchAndLock()
                self.image_capture.capture("images/")
                self.cam.unlock()
            except:
                print("No camera in viewfinder.")
# Run program
if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = Camera()
    sys.exit(app.exec_())
```

Your GUI should look similar to the one in Figure 12-3.
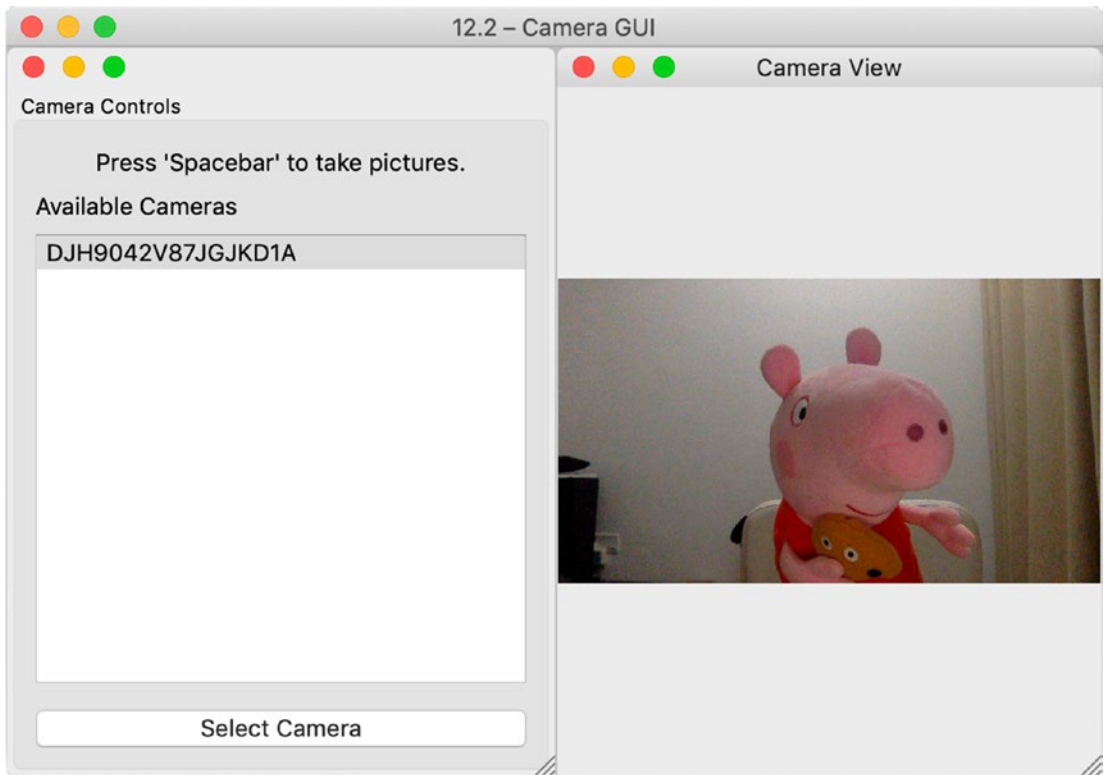
Figure 12-3. *Camera GUI is composed of multiple windows that allow the user to select available cameras and view the camera's viewfinder*

Explanation

For this project, we are going to import some new classes. From the QtWidgets module, the QMdiArea and QMdiSubWindow classes are used to create the MDI windows.

The **QtMultimedia** module provides access to a number of multimedia tools including audio, video, and camera capabilities. The QCamera class provides the interface to work with camera devices. QCameraInfo supplies information about available cameras. QCameraImageCapture is used for recording media.

From the **QtMultimediaWidgets** module, the QCameraViewfinder class sets up the camera viewfinder widget. In photography, the viewfinder is used for focusing and viewing the subject being photographed.

This application contains two subwindows, one for displaying the viewfinder and the other for listing the available cameras that you can choose from in a QListWidget object. In the setupWindows() method, the labels, list widget, and push button are

arranged inside of a QGroupBox widget. The user can select a camera from the list. The Select Camera button emits a signal that is connected to the selectCamera() slot. Next, the QMdiArea object, mdi_area, that is used as a container for the subwindows is created. This will be the central widget for the main window.

Child windows are instances of QMdiSubWindow. The subwindows inside of mdi_area are created in relation to each other. In this project, they are arranged as tiles using tileSubWindows(). Another option is to lay them out using a cascaded style.

```
mdi_area.cascadeSubWindows()
```

Tip   A menubar could also be added to the main window that controls the subwindows. For example, subwindows could be set as checkable in order to close or reopen them. Or a menu item could allow the user to switch between tiled or cascaded windows.

If the user clicks the push button and an available camera is selected, then the setupCamera() method is called. Refer to the comments in the code to learn how to set up the viewfinder. This method is adapted from the Qt document web site.[1]

Using QCameraImageCapture(), the user is also able to take pictures of the viewfinder. Image capturing is handled by the keyPressEvent(). When the spacebar is pressed, a picture is taken and saved to the "images/" folder. The folder will be created if it does not already exist.

Project 12.3 – Simple Clock GUI

PyQt5 also provides classes for dealing with dates, **QDate**, or time, **QTime**. The **QDateTime** class supplies functions for working with both dates and time. All three of these classes include methods for handling time-related features.

Let's take a brief look at the QDateTime class. The following snippet of code creates an instance of QDateTime that returns the current date and time using the currentDateTime() method:

```
date_time = QDateTime.currentDateTime()
print(date_time.toString(Qt.DefaultLocaleLongDate)))
```

---

[1]https://doc.qt.io/qt-5/qcameraviewfinder.html

The current date and time is printed to the screen with the following format (set using `Qt.DefaultLocaleLongDate`):

```
February 15, 2020 2:32:31 PM CST
```

There are also other formats, including shorter formats, ISO 8601 format, or UTC format. The `toString()` method returns the date and time as a string. QDateTime also handles daylight saving time, different time zones, and the manipulation of times and dates such as adding or subtracting months, days, or hours.

If you only need to work with the individual dates and times, QDate and QTime also provide similar functions as you shall see in Listing 12-3.

Listing 12-3. Code for the clock GUI

```python
# clock.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
    QVBoxLayout)
from PyQt5.QtCore import Qt, QDate, QTime, QTimer

class DisplayTime(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 250, 100)
        self.setWindowTitle('12.3 - QDateTime Example')
        self.setStyleSheet("background-color: black")

        self.setupWidgets()
```

```python
        # Create timer object
        timer = QTimer(self)
        timer.timeout.connect(self.updateDateTime)
        timer.start(1000)

        self.show()

    def setupWidgets(self):
        """
        Set up labels that will display current date and time.
        """
        current_date, current_time = self.getDateTime()

        self.date_label = QLabel(current_date)
        self.date_label.setStyleSheet("color: white; font: 16px Courier")
        self.time_label = QLabel(current_time)
        self.time_label.setStyleSheet("""color: white;
                                          border-color: white;
                                          border-width: 2px;
                                          border-style: solid;
                                          border-radius: 4px;
                                          padding: 10px;
                                          font: bold 24px Courier""")

        # Create layout and add widgets
        v_box = QVBoxLayout()
        v_box.addWidget(self.date_label, alignment=Qt.AlignCenter)
        v_box.addWidget(self.time_label, alignment=Qt.AlignCenter)

        self.setLayout(v_box)

    def getDateTime(self):
        """
        Returns current date and time.
        """
        date = QDate.currentDate().toString(Qt.DefaultLocaleLongDate)

        time = QTime.currentTime().toString("hh:mm:ss AP")
        return date, time
```

```
    def updateDateTime(self):
        """

        Slot that updates date and time values.
        """

        date = QDate.currentDate().toString(Qt.DefaultLocaleLongDate)
        time = QTime.currentTime().toString("hh:mm:ss AP")

        self.date_label.setText(date)
        self.time_label.setText(time)
        return date, time

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = DisplayTime()
    sys.exit(app.exec_())
```

The clock application can be seen in Figure 12-4.
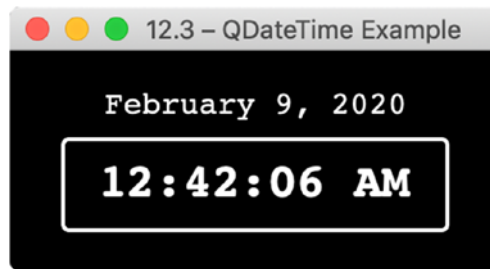
Figure 12-4. *The clock GUI displaying the current calendar date and clock time*

Explanation

Start by importing the necessary modules, including QDate, QTime, and QTimer, from the QtCore module. The QTimer class will be used to create a timer object to keep track of the time that has passed and update the labels that hold the date and time accordingly. The timer is set up in initializeUI(), and its timeout() signal is connected to the updateDateTime() slot. The timeout() signal is emitted every second.

In order to get the current date and time, the values are retrieved using the `currentDate()` and `currentTime()` methods in the `getDateTime()` method. These are then returned and set as the `current_date` and `current_time`.

```
current_date, current_time = self.getDateTime()
```

While the `date` is set to use the `Qt.DefaultLocaleLongDate` format, the `time` uses a sequence of characters to create a format string that displays hours (`hh`), minutes (`mm`), seconds (`ss`), and AM or PM (`AP`).

```
time = QTime.currentTime().toString("hh:mm:ss AP")
```

The labels that will display the date and time are then instantiated, styled, and added to the layout. The values of the labels are updated using the `updateDateTime()` method.

Project 12.4 – Calendar GUI

This project takes a look at how to set up the **QCalendarWidget** class and use a few of its functions. PyQt makes adding a monthly calendar to your applications rather effortless. The code is provided in Listing 12-4 and the calendar can be seen in Figure 12-5.

The QCalendarWidget class provides a calendar that already has a number of other useful widgets and functions built-in. For example, the calendar already includes a horizontal header that includes widgets for changing the month and the year and a vertical header that displays the week number. The class also includes signals that are emitted whenever the dates, months, and years on the calendar are changed.

The **QDateEdit** widget is used in this application to restrict the values a user can select to within a certain range, specified by minimum and maximum values.

Listing 12-4. The calendar GUI code

```
# calendar.py
# Import necessary modules
import sys
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel,
QCalendarWidget, QDateEdit, QGroupBox, QHBoxLayout, QGridLayout)
from PyQt5.QtCore import Qt, QDate
from PyQt5.QtGui import QFont
```

```
style_sheet = """
    QLabel{
        padding: 5px;
        font: 18px
    }

    QLabel#DateSelected{
        font: 24px
    }

    QGroupBox{
        border: 2px solid gray;
        border-radius: 5px;
        margin-top: 1ex;
        font: 14px
    }
"""

class CalendarGUI(QWidget):

    def __init__(self):
        super().__init__()
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setMinimumSize(500, 400)
        self.setWindowTitle('12.4 - Calendar GUI')

        self.createCalendar()

        self.show()

    def createCalendar(self):
        """
        Set up calendar, others widgets and layouts for main window.
        """
```

```python
        self.calendar = QCalendarWidget()
        self.calendar.setGridVisible(True)
        self.calendar.setMinimumDate(QDate(1900, 1, 1))
        self.calendar.setMaximumDate(QDate(2200, 1, 1))

        # Connect to newDateSelection slot when currently selected date is
          changed
        self.calendar.selectionChanged.connect(self.newDateSelection)

        current = QDate.currentDate().toString(Qt.DefaultLocaleLongDate)
        self.current_label = QLabel(current)
        self.current_label.setObjectName("DateSelected")

        # Create current, minimum, and maximum QDateEdit widgets
        min_date_label = QLabel("Minimum Date:")
        self.min_date_edit = QDateEdit()
        self.min_date_edit.setDisplayFormat("MMM d yyyy")
        self.min_date_edit.setDateRange(self.calendar.minimumDate(), self.
        calendar.maximumDate())
        self.min_date_edit.setDate(self.calendar.minimumDate())
        self.min_date_edit.dateChanged.connect(self.minDatedChanged)

        current_date_label = QLabel("Current Date:")
        self.current_date_edit = QDateEdit()
        self.current_date_edit.setDisplayFormat("MMM d yyyy")
        self.current_date_edit.setDate(self.calendar.selectedDate())
        self.current_date_edit.setDateRange(self.calendar.minimumDate(),
        self.calendar.maximumDate())
        self.current_date_edit.dateChanged.connect(self.
        selectionDateChanged)

        max_date_label = QLabel("Maximum Date:")
        self.max_date_edit = QDateEdit()
        self.max_date_edit.setDisplayFormat("MMM d yyyy")
        self.max_date_edit.setDateRange(self.calendar.minimumDate(), self.
        calendar.maximumDate())
        self.max_date_edit.setDate(self.calendar.maximumDate())
        self.max_date_edit.dateChanged.connect(self.maxDatedChanged)
```

```
        # Add widgets to group box and add to grid layout
        dates_gb = QGroupBox("Set Dates")
        dates_grid = QGridLayout()
        dates_grid.addWidget(self.current_label, 0, 0, 1, 2,
        Qt.AlignAbsolute)
        dates_grid.addWidget(min_date_label, 1, 0)
        dates_grid.addWidget(self.min_date_edit, 1, 1)
        dates_grid.addWidget(current_date_label, 2, 0)
        dates_grid.addWidget(self.current_date_edit, 2, 1)
        dates_grid.addWidget(max_date_label, 3, 0)
        dates_grid.addWidget(self.max_date_edit, 3, 1)
        dates_gb.setLayout(dates_grid)

        # Create and set main window's layout
        main_h_box = QHBoxLayout()
        main_h_box.addWidget(self.calendar)
        main_h_box.addWidget(dates_gb)

        self.setLayout(main_h_box)

    def selectionDateChanged(self, date):
        """
        Update the current_date_edit when the calendar's selected date
        changes.
        """
        self.calendar.setSelectedDate(date)

    def minDatedChanged(self, date):
        """
        Update the calendar's minimum date.
        Update max_date_edit to avoid conflicts with maximum and minimum
        dates.
        """
        self.calendar.setMinimumDate(date)
        self.max_date_edit.setDate(self.calendar.maximumDate())
```

```
    def maxDatedChanged(self, date):
        """
        Update the calendar's maximum date.
        Update min_date_edit to avoid conflicts with minimum and maximum
        dates.
        """
        self.calendar.setMaximumDate(date)
        self.min_date_edit.setDate(self.calendar.minimumDate())

    def newDateSelection(self):
        """
        Update date in current_label and current_date_edit widgets when
        user selects a new date.
        """
        date = self.calendar.selectedDate().toString
        (Qt.DefaultLocaleLongDate)
        self.current_date_edit.setDate(self.calendar.selectedDate())
        self.current_label.setText(date)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    app.setStyleSheet(style_sheet)
    window = CalendarGUI()
    sys.exit(app.exec_())
```

The look of your calendar will greatly depend upon the platform that you are using to run the application. An example of the calendar on MacOS can be seen in Figure 12-5.
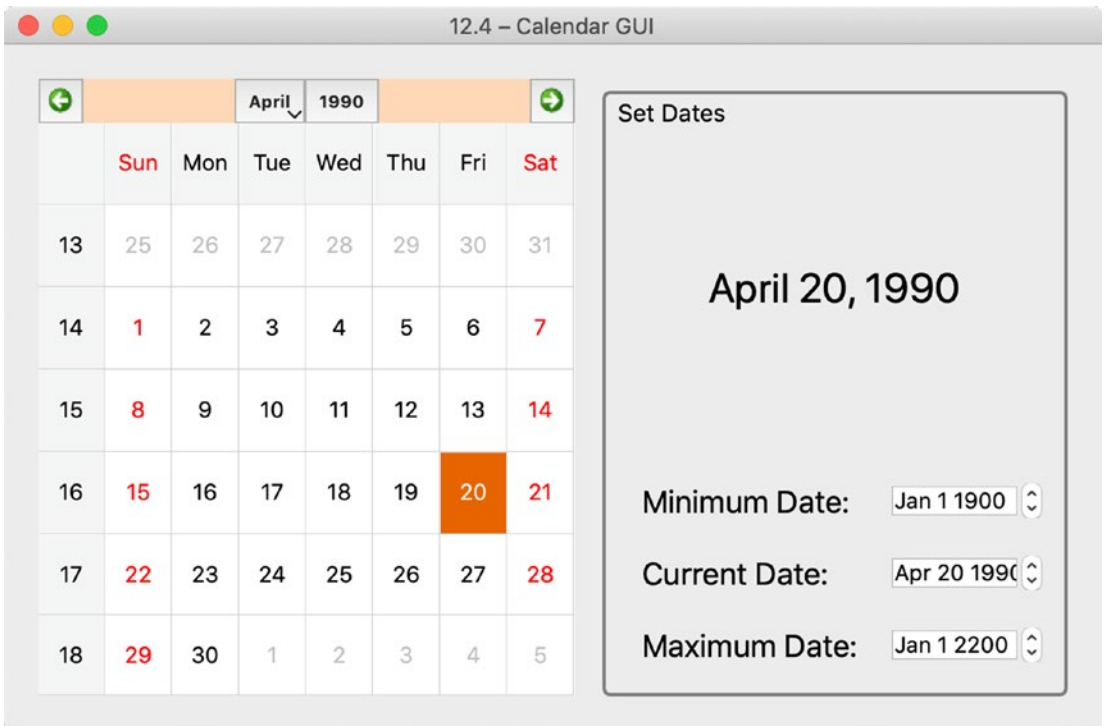
Figure 12-5. *The calendar GUI that displays the calendar, the current date, and the widgets that allow the user to search for dates within a specified time range*

Explanation

After importing the modules needed for the calendar GUI, the styles for the QLabel and QGroupBox widgets are prepared using style_sheet.

Creating an instance of QCalendarWidget is very simple.

```
self.calendar = QCalendarWidget()
```

Next, we set a few of the calendar object's parameters. Setting setGridVisible() to True will make the grid lines visible. In order to specify the date range that a user can select in the calendar, we set the minimum and maximum date values.

```
self.calendar.setMinimumDate(QDate(1900, 1, 1))
self.calendar.setMaximumDate(QDate(2200, 1, 1))
```

Whenever a date is selected in the `calendar` widget, it emits a `selectionChanged()` signal. This signal is connected to the `newDateSelection()` slot that updates the date on the `current_label` and in the `current_date_edit` widget. Selecting a value in the `current_date_edit` widget will also change the other values.

The QCalendarWidget class also has a number of functions that allow you to configure its behaviors and appearance. For this project, we create three `QDateEdit` widgets that will allow the user to change the minimum and maximum values for the date range, as well as the current date selected in the calendar.

A displayed format for the date in the `QDateEdit` widget can be set using `setDisplayFormat()`. The date edit objects are also given a date range using `setDateRange()`. The following code is an example of how to set the `min_date_edit` widget's date range by using ranges set earlier for the `calendar` object:

```
self.min_date_edit.setDateRange(self.calendar.minimumDate(), self.calendar.
maximumDate())
```

When a date is changed in a date edit widget, it generates a `dateChanged()` signal. Each one of the `QDateEdit` widgets is connected to a corresponding slot that will update the calendar's minimum, maximum, or current date values depending upon which date edit widget is changed. The method for changing the dates is adapted from the Qt document web site.[2]

Finally, the label and date edit widgets are arranged in a `QGroupBox`.

Project 12.5 – Hangman GUI

PyQt can be used to create a variety of different kinds of applications. Throughout this book, we have looked at quite a few ideas for building GUIs. For this next project, we will be taking a look at how to use QPainter and a few other classes to build a game – Hangman. While Hangman is a simple game to play, it can be used to teach a few of the fundamental concepts for using PyQt to create games. The code is presented in Listing 12-5 and the interface can be seen in Figure 12-6.

---

[2]https://doc.qt.io/qt-5/qtwidgets-widgets-calendarwidget-example.html

For this application, the player can select from one of the twenty-six English letters to guess a letter in an unknown word. As each letter is chosen, they will become disabled in the window. If the letter is correct, it will be revealed to the player. Otherwise, a part of the hangman figure's body is drawn on the screen. If all of the letters are correctly guessed, then the player wins. There are a total of six turns.

Whether or not the player wins or loses, a dialog will be displayed to inform the player and allow them to quit or to continue playing.

Listing 12-5. This is the code for the Hangman GUI

```
# hangman.py
# Import necessary modules
import sys, random
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
QPushButton, QLabel, QFrame, QButtonGroup, QHBoxLayout, QVBoxLayout,
QMessageBox, QSizePolicy)
from PyQt5.QtCore import Qt, QRect, QLine
from PyQt5.QtGui import QPainter, QPen, QBrush, QColor

style_sheet = """
    QWidget{
        background-color: #FFFFFF
    }

    QLabel#Word{
        font: bold 20px;
        qproperty-alignment: AlignCenter
    }

    QPushButton#Letters{
        background-color: #1FAEDE;
        color: #D2DDE1;
        border-style: solid;
        border-radius: 3px;
        border-color: #38454A;
        font: 28px
    }
```

```
    QPushButton#Letters:pressed{
        background-color: #C86354;
        border-radius: 4px;
        padding: 6px;
        color: #DFD8D7
    }

    QPushButton#Letters:disabled{
        background-color: #BBC7CB
    }
"""

# The hangman is drawn on a QLabel object, rather than
# on the main window. This class handles the drawing.
class DrawingLabel(QLabel):

    def __init__(self, parent):
        super().__init__(parent)

        # Variables for positioning drawings
        self.height = 200
        self.width = 300

        # Variables used to keep track of incorrect guesses
        self.incorrect_letter = False
        self.incorrect_turns = 0

        # List to store body parts
        self.part_list = []

    def drawHangmanBackground(self, painter):
        """
        Draw the gallows.
        """
        painter.setBrush(QBrush(QColor("#000000")))
        # drawRect(x, y, width, height)
        painter.drawRect((self.width / 2) - 40, self.height, 150, 4)
        painter.drawRect(self.width / 2, 0, 4, 200)
```

```python
        painter.drawRect(self.width / 2, 0, 60, 4)
        painter.drawRect((self.width / 2) + 60, 0, 4, 40)

    def drawHangmanBody(self, painter):
        """
        Create and draw body parts for hangman.
        """
        if "head" in self.part_list:
            head = QRect((self.width / 2) + 42, 40, 40, 40)
            painter.setPen(QPen(QColor("#000000"), 3))
            painter.setBrush(QBrush(QColor("#FFFFFF")))
            painter.drawEllipse(head)
        if "body" in self.part_list:
            body = QRect((self.width / 2) + 60, 80, 2, 55)
            painter.setBrush(QBrush(QColor("#000000")))
            painter.drawRect(body)
        if "right_arm" in self.part_list:
            right_arm = QLine((self.width / 2) + 60, 85,
                (self.width / 2) + 50, (self.height / 2) + 30)
            pen = QPen(Qt.black, 3, Qt.SolidLine)
            painter.setPen(pen)
            painter.drawLine(right_arm)
        if "left_arm" in self.part_list:
            left_arm = QLine((self.width / 2) + 62, 85,
                (self.width / 2) + 72, (self.height / 2) + 30)
            painter.drawLine(left_arm)
        if "right_leg" in self.part_list:
            right_leg = QLine((self.width / 2) + 60, 135,
                (self.width / 2) + 50, (self.height / 2) + 75)
            painter.drawLine(right_leg)
        if "left_leg" in self.part_list:
            left_leg = QLine((self.width / 2) + 62, 135,
                (self.width / 2) + 72, (self.height / 2) + 75)
            painter.drawLine(left_leg)
```

```python
        # Reset variable
        self.incorrect_letter = False

    def paintEvent(self, event):
        """
        Construct the QPainter and handle painting events.
        """
        painter = QPainter(self)

        self.drawHangmanBackground(painter)

        if self.incorrect_letter == True:
            self.drawHangmanBody(painter)

        painter.end()

class Hangman(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setFixedSize(400, 500)
        self.setWindowTitle('12.5 - Hangman GUI')

        self.newGame()

        self.show()

    def newGame(self):
        """
        Create new Hangman game.
        """
        self.setupHangmanBoard()
        self.setupWord()
        self.setupBoard()
```

```python
def setupHangmanBoard(self):
    """
    Set up label object to display hangman.
    """
    self.hangman_label = DrawingLabel(self)
    self.hangman_label.setSizePolicy(QSizePolicy.Expanding,
    QSizePolicy.Expanding)

def setupWord(self):
    """
    Open words file and choose random word.
    Create labels that will display '_' depending
    upon length of word.
    """
    words = self.openFile()
    self.chosen_word = random.choice(words).upper()
    #print(self.chosen_word)

    # Keep track of correct guesses
    self.correct_counter = 0

    # Keep track of label objects.
    # Is used for updating the text on the labels
    self.labels = []

    word_h_box = QHBoxLayout()

    for letter in self.chosen_word:
        self.letter_label = QLabel("___")
        self.labels.append(self.letter_label)
        self.letter_label.setObjectName("Word")
        word_h_box.addWidget(self.letter_label)

    self.word_frame = QFrame()
    self.word_frame.setLayout(word_h_box)

def setupBoard(self):
    """
    Set up objects and layouts for keyboard and main window.
    """
```

```python
        top_row_list = ["A", "B", "C", "D", "E", "F", "G", "H"]
        mid_row_list = ["I", "J", "K", "L", "M", "N", "O", "P", "Q"]
        bot_row_list = ["R", "S", "T", "U", "V", "W", "X", "Y", "Z"]

        # Create button group to keep track of letters
        self.keyboard_bg = QButtonGroup()

        # Set up keys in the top row
        top_row_h_box = QHBoxLayout()

        for letter in top_row_list:
            button = QPushButton(letter)
            button.setObjectName("Letters")
            top_row_h_box.addWidget(button)
            self.keyboard_bg.addButton(button)

        top_frame = QFrame()
        top_frame.setLayout(top_row_h_box)

        # Set up keys in the middle row
        mid_row_h_box = QHBoxLayout()

        for letter in mid_row_list:
            button = QPushButton(letter)
            button.setObjectName("Letters")
            mid_row_h_box.addWidget(button)
            self.keyboard_bg.addButton(button)

        mid_frame = QFrame()
        mid_frame.setLayout(mid_row_h_box)

        # Set up keys in the bottom row
        bot_row_h_box = QHBoxLayout()

        for letter in bot_row_list:
            button = QPushButton(letter)
            button.setObjectName("Letters")
            bot_row_h_box.addWidget(button)
            self.keyboard_bg.addButton(button)
```

```python
        bot_frame = QFrame()
        bot_frame.setLayout(bot_row_h_box)

        # Connect buttons in button group to slot
        self.keyboard_bg.buttonClicked.connect(self.buttonPushed)

        keyboard_v_box = QVBoxLayout()
        keyboard_v_box.addWidget(top_frame)
        keyboard_v_box.addWidget(mid_frame)
        keyboard_v_box.addWidget(bot_frame)

        keyboard_frame = QFrame()
        keyboard_frame.setLayout(keyboard_v_box)

        # Create main layout and add widgets
        main_v_box = QVBoxLayout()
        main_v_box.addWidget(self.hangman_label)
        main_v_box.addWidget(self.word_frame)
        main_v_box.addWidget(keyboard_frame)

        # Create central widget for main window
        central_widget = QWidget()
        central_widget.setLayout(main_v_box)

        self.setCentralWidget(central_widget)

    def buttonPushed(self, button):
        """
        Handle buttons from the button group and game logic.
        """
        button.setEnabled(False)

        body_parts_list = ["head", "body", "right_arm",
            "left_arm", "right_leg", "left_leg"]

        # When the user guesses incorrectly and the number of incorrect
        # turns is not equal to 6 (the number of body parts).
        if button.text() not in self.chosen_word and self.hangman_label.
        incorrect_turns <= 5:
            self.hangman_label.incorrect_turns += 1
```

```
            index = self.hangman_label.incorrect_turns - 1
            self.hangman_label.part_list.append(body_parts_list[index])
            self.hangman_label.incorrect_letter = True
        # When a correct letter is chosen, update labels and
        # correct counter.
        elif button.text() in self.chosen_word and self.hangman_label.
        incorrect_turns <= 5:
            self.hangman_label.incorrect_letter = True
            for i in range(len(self.chosen_word)):
                if self.chosen_word[i] == button.text():
                    self.labels[i].setText(button.text())
                    self.correct_counter += 1

        # Call update before checking winning conditions
        self.update()

        # User wins when the number of correct letters equals
        # the length of the word.
        if self.correct_counter == len(self.chosen_word):
            self.displayDialogs("win")

        # Game over if number of incorrect turns equals
        # the number of body parts. Reveal word to user.
        if self.hangman_label.incorrect_turns == 6:
            for i in range(len(self.chosen_word)):
                self.labels[i].setText(self.chosen_word[i])
            self.displayDialogs("game_over")

    def openFile(self):
        """
        Open words.txt file.
        """
        try:
            with open("files/words.txt", 'r') as f:
                word_list = f.read().splitlines()
                return word_list
```

```
        except FileNotFoundError:
            print("File Not Found.")
            ex_list = ["nofile"]
            return ex_list

    def displayDialogs(self, text):
        """
        Display win and game over dialog boxes.
        """
        if text == "win":
            message = QMessageBox().question(self, "Win!",
                "You Win!\nNEW GAME?", QMessageBox.Yes | QMessageBox.No,
                QMessageBox.No)
        elif text == "game_over":
            message = QMessageBox().question(self, "Game Over",
                "Game Over\nNEW GAME?", QMessageBox.Yes | QMessageBox.No,
                QMessageBox.No)

        if message == QMessageBox.No:
            self.close()
        else:
            self.newGame()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    app.setStyleSheet(style_sheet)
    window = Hangman()
    sys.exit(app.exec_())
```

The finished hangman GUI can be seen in Figure 12-6.

361
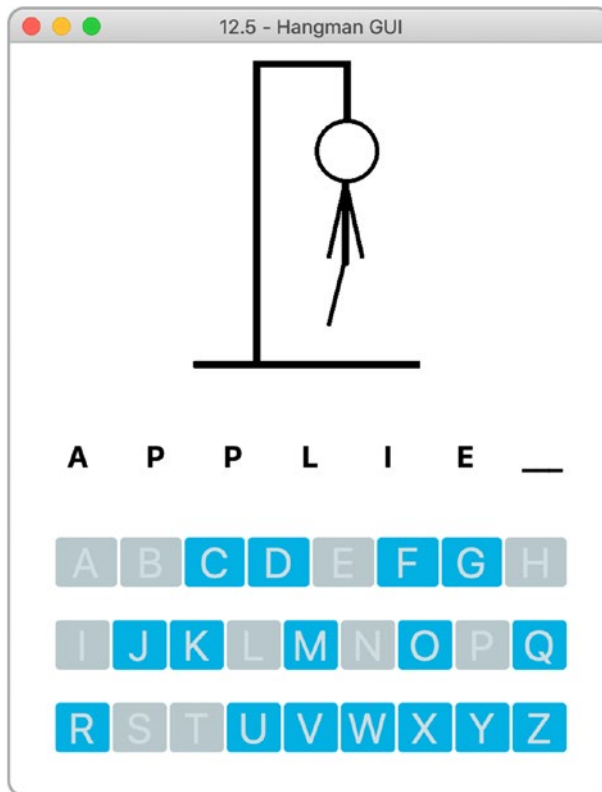
Figure 12-6. *The Hangman application. Can you save him?*

Explanation

A variety of classes are used in the Hangman GUI, including different widgets from QtWidgets, as well as classes used for drawing from QtCore and QtGui. We then create a style sheet to set the style properties of the widgets and to handle the situation of how the buttons look when they are pressed.

This program contains two classes, DrawingLabel and Hangman.

Creating the Drawing Class

The DrawingLabel class inherits from QLabel and handles the different paint events that will be drawn on the label object in the main window. The paintEvent() function is called in a class that inherits from QLabel so that way the paint events occur on the label and are not covered up by the main window.

In order to use this class, an instance is created in the Hangman class:

```
self.hangman_label = DrawingLabel(self)
```

The paintEvent() function sets up QPainter and handles the two painting methods, drawHangmanBackground(), which draws the gallows of the Hangman game onto the label, and drawHangmanBody(), which only draws the body parts if they are contained in the part_list.

Creating the Main Window Class

The Hangman class starts by initializing the GUI window and calling the newGame() method. First, the Hangman board is created as an instance of the DrawingLabel class. Then, setupBoard() selects a random word from the words.txt file. The labels that will represent the letters of the chosen word are replaced with underscore characters, appended to the labels list, and added to the horizontal layout of the word_frame object.

Finally, we need to set up the keyboard push buttons, layouts, and the game logic in setupBoard(). Three rows of push buttons that represent the letters of the alphabet are controlled by one QButtonGroup object, keyboard_bg. When one button is pushed, it generates a signal that calls the buttonPushed slot.

When a push button is pressed, it is disabled.

```
button.setEnabled(False)
```

The list of body parts contains the six body part names. If the player guesses an incorrect letter, the name is appended to the part_list and checked for in the drawHangmanBody() function. Using this method ensures that all necessary parts are drawn with their different styles when paintEvent() is called. Otherwise, the labels are updated to display the correct letters in the appropriate positions if the player guesses correctly.

If the player wins or loses, a QMessageBox will appear and allow the user to close the application or continue. If Yes is selected, newGame() is called.

Project 12.6 – Web Browser GUI

A web browser is a graphical user interface that allows access to information on the World Wide Web (Listing 12-6). A user can enter a Uniform Resource Locator (URL) into an address bar and request content for a web site from a web server to be displayed on their local device, including text, image, and video data. URLs are generally prefixed with `http`, a protocol used for fetching and transmitting requested web pages, or `https`, for encrypted communication between browsers and web sites.

Qt provides quite a few classes for network communication, WebSockets, support for accessing the World Wide Web, and more. This project introduces PyQt's classes for web integration into GUIs.

For the following project, we will take a look at QtWebEngine, specifically the **QtWebEngineWidgets** module for creating widget-based web applications. QtWebEngine provides a web browser engine that can be used to embed web content into your applications. The QtWebEngine module uses Chromium as its back end. Chromium is open source software from Google that can be used to create web browsers.

The web browser GUI serves as a framework for creating your own web browser and includes the following features:

- Ability to open multiple windows and tabs, either by using the application's menu or shortcut hot keys

- A navigation bar that is made up of back, forward, refresh, stop, and home buttons and the address bar for entering URLs

- The web engine view widget created using `QWebEngineView`

- A status bar

- A progress bar that relays feedback to the user about loading web pages

---

Note   When you are running this program, if you get an error message stating "No module named: PyQt5.QtWebEngineWidgets", then you need to install the `QtWebEngineWidgets` module. To solve this problem, enter the following command into the command line: `pip3 install PyQtWebEngine`.

Listing 12-6. Web browser GUI code

```python
# web_browser.py
# Import necessary modules
import os, sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QLabel,
QLineEdit, QDesktopWidget, QTabWidget, QHBoxLayout, QVBoxLayout, QAction,
QToolBar, QProgressBar, QStatusBar)
from PyQt5.QtCore import QSize, QUrl
from PyQt5.QtGui import QIcon
from PyQt5.QtWebEngineWidgets import QWebEngineView

style_sheet = """
    QTabWidget:pane{
        border: none
    }
"""

class WebBrowser(QMainWindow):
    def __init__(self):
        super().__init__()

        # Create lists that will keep track of the new windows,
        # tabs, and URLs
        self.window_list = []
        self.list_of_web_pages = []
        self.list_of_urls = []

        self.initializeUI()

    def initializeUI(self):
        self.setMinimumSize(300, 200)
        self.setWindowTitle("12.6 - Web Browser")
        self.setWindowIcon(QIcon(os.path.join('images', 'pyqt_logo.png')))

        self.positionMainWindow()
```

```python
        self.createMenu()
        self.createToolbar()
        self.createTabs()

        self.show()

    def createMenu(self):
        """
        Set up the menu bar.
        """
        new_window_act = QAction('New Window', self)
        new_window_act.setShortcut('Ctrl+N')
        new_window_act.triggered.connect(self.openNewWindow)

        new_tab_act = QAction('New Tab', self)
        new_tab_act.setShortcut('Ctrl+T')
        new_tab_act.triggered.connect(self.openNewTab)

        quit_act = QAction("Quit Browser", self)
        quit_act.setShortcut('Ctrl+Q')
        quit_act.triggered.connect(self.close)

        # Create the menu bar
        menu_bar = self.menuBar()
        menu_bar.setNativeMenuBar(False)

        # Create file menu and add actions
        file_menu = menu_bar.addMenu('File')
        file_menu.addAction(new_window_act)
        file_menu.addAction(new_tab_act)
        file_menu.addSeparator()
        file_menu.addAction(quit_act)

        self.status_bar = QStatusBar()
        self.setStatusBar(self.status_bar)

    def createToolbar(self):
        """
        Set up the navigation toolbar.
        """
```

```
tool_bar = QToolBar("Address Bar")
tool_bar.setIconSize(QSize(30, 30))
self.addToolBar(tool_bar)

# Create toolbar actions
back_page_button = QAction(QIcon(os.path.join('icons',
'back.png')), "Back", self)
back_page_button.triggered.connect(self.backPageButton)

forward_page_button = QAction(QIcon(os.path.join('icons',
'forward.png')), "Forward", self)
forward_page_button.triggered.connect(self.forwardPageButton)

refresh_button = QAction(QIcon(os.path.join('icons',
'refresh.png')), "Refresh", self)
refresh_button.triggered.connect(self.refreshButton)

home_button = QAction(QIcon(os.path.join('icons', 'home.png')),
"Home", self)
home_button.triggered.connect(self.homeButton)

stop_button = QAction(QIcon(os.path.join('icons', 'stop.png')),
"Stop", self)
stop_button.triggered.connect(self.stopButton)

# Set up the address bar
self.address_line = QLineEdit()
# addAction() is used here to merely display the icon in the line edit
self.address_line.addAction(QIcon('icons/search.png'), QLineEdit.
LeadingPosition)
self.address_line.setPlaceholderText("Enter website address")
self.address_line.returnPressed.connect(self.searchForUrl)

tool_bar.addAction(home_button)
tool_bar.addAction(back_page_button)
tool_bar.addAction(forward_page_button)
tool_bar.addAction(refresh_button)
tool_bar.addWidget(self.address_line)
tool_bar.addAction(stop_button)
```

```python
def createTabs(self):
    """
    Create the QTabWidget object and the different pages.
    Handle when a tab is closed.
    """
    self.tab_bar = QTabWidget()
    self.tab_bar.setTabsClosable(True) # Add close buttons to tabs
    self.tab_bar.setTabBarAutoHide(True) # Hides tab bar when less than
    2 tabs
    self.tab_bar.tabCloseRequested.connect(self.closeTab)

    # Create tab
    self.main_tab = QWidget()
    self.tab_bar.addTab(self.main_tab, "New Tab")

    # Call method that sets up each page
    self.setupTab(self.main_tab)

    self.setCentralWidget(self.tab_bar)

def setupWebView(self):
    """
    Create the QWebEngineView object that is used to view
    web docs. Set up the main page, and handle web_view signals.
    """
    web_view = QWebEngineView()
    web_view.setUrl(QUrl("https://google.com"))

    # Create page loading progress bar that is displayed in
    # the status bar.
    self.page_load_pb = QProgressBar()
    self.page_load_label = QLabel()
    web_view.loadProgress.connect(self.updateProgressBar)

    # Display URL in address bar
    web_view.urlChanged.connect(self.updateUrl)
```

```
        ok = web_view.loadFinished.connect(self.updateTabTitle)
        if ok:
            # Web page loaded
            return web_view
        else:
            print("The request timed out.")

    def setupTab(self, tab):
        """
        Create individual tabs and widgets. Add the tab's url and web view
        to the appropriate list.
        Update the address bar if the user switches tabs.
        """
        # Create the web view that will be displayed on the page.
        self.web_page = self.setupWebView()

        tab_v_box = QVBoxLayout()
        # Sets the left, top, right, and bottom margins to use around the
          layout.
        tab_v_box.setContentsMargins(0,0,0,0)
        tab_v_box.addWidget(self.web_page)

        # Append new web_page and URL to the appropriate lists
        self.list_of_web_pages.append(self.web_page)
        self.list_of_urls.append(self.address_line)
        self.tab_bar.setCurrentWidget(self.web_page)

        # If user switches pages, update the URL in the address to
        # reflect the current page.
        self.tab_bar.currentChanged.connect(self.updateUrl)

        tab.setLayout(tab_v_box)

    def openNewWindow(self):
        """
        Create new instance of the WebBrowser class.
        """
        new_window = WebBrowser()
```

```python
        new_window.show()
        self.window_list.append(new_window)

    def openNewTab(self):
        """
        Create new tabs.
        """
        new_tab = QWidget()
        self.tab_bar.addTab(new_tab, "New Tab")
        self.setupTab(new_tab)

        # Update the tab_bar index to keep track of the new tab.
        # Load the URL for the new page.
        tab_index = self.tab_bar.currentIndex()
        self.tab_bar.setCurrentIndex(tab_index + 1)

        self.list_of_web_pages[self.tab_bar.currentIndex()].
        load(QUrl("https://google.com"))

    def updateProgressBar(self, progress):
        """
        Update progress bar in status bar.
        This provides feedback to the user that the page is still loading.
        """
        if progress < 100:
            self.page_load_pb.setVisible(progress)
            self.page_load_pb.setValue(progress)
            self.page_load_label.setVisible(progress)
            self.page_load_label.setText("Loading Page... ({}/100)".
            format(str(progress)))
            self.status_bar.addWidget(self.page_load_pb)
            self.status_bar.addWidget(self.page_load_label)
        else:
            self.status_bar.removeWidget(self.page_load_pb)
            self.status_bar.removeWidget(self.page_load_label)

    def updateTabTitle(self):
        """
```

```
    Update the title of the tab to reflect the website.
    """
    tab_index = self.tab_bar.currentIndex()
    title = self.list_of_web_pages[self.tab_bar.currentIndex()].page().
    title()
    self.tab_bar.setTabText(tab_index, title)

def updateUrl(self):
    """
    Update the url in the address to reflect the current page being
    displayed.
    """
    url = self.list_of_web_pages[self.tab_bar.currentIndex()].page().url()
    formatted_url = QUrl(url).toString()
    self.list_of_urls[self.tab_bar.currentIndex()].setText(formatted_url)

def searchForUrl(self):
    """
    Make a request to load a url.
    """
    url_text = self.list_of_urls[self.tab_bar.currentIndex()].text()

    # Append http to URL
    url = QUrl(url_text)
    if url.scheme() == "":
        url.setScheme("http")

    # Request URL
    if url.isValid():
        self.list_of_web_pages[self.tab_bar.currentIndex()].page().
        load(url)
    else:
        url.clear()

def backPageButton(self):
    tab_index = self.tab_bar.currentIndex()
    self.list_of_web_pages[tab_index].back()
```

```python
def forwardPageButton(self):
    tab_index = self.tab_bar.currentIndex()
    self.list_of_web_pages[tab_index].forward()

def refreshButton(self):
    tab_index = self.tab_bar.currentIndex()
    self.list_of_web_pages[tab_index].reload()

def homeButton(self):
    tab_index = self.tab_bar.currentIndex()
    self.list_of_web_pages[tab_index].setUrl(QUrl("https://google.
    com"))

def stopButton(self):
    tab_index = self.tab_bar.currentIndex()
    self.list_of_web_pages[tab_index].stop()

def closeTab(self, tab_index):
    """
    This signal is emitted when the close button on a tab is clicked.
    The index is the index of the tab that should be removed.
    """
    self.list_of_web_pages.pop(tab_index)
    self.list_of_urls.pop(tab_index)

    self.tab_bar.removeTab(tab_index)

def positionMainWindow(self):
    """
    Use QDesktopWidget class to access information about your screen
    and use it to position the application window when starting a new
    application.
    """
    desktop = QDesktopWidget().screenGeometry()
    screen_width = desktop.width()
    screen_height = desktop.height()
    self.setGeometry(0, 0, screen_width, screen_height)
```

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    app.setStyleSheet(style_sheet)
    window = WebBrowser()
    app.exec_()
```

Your application should look similar to Figure 12-7.
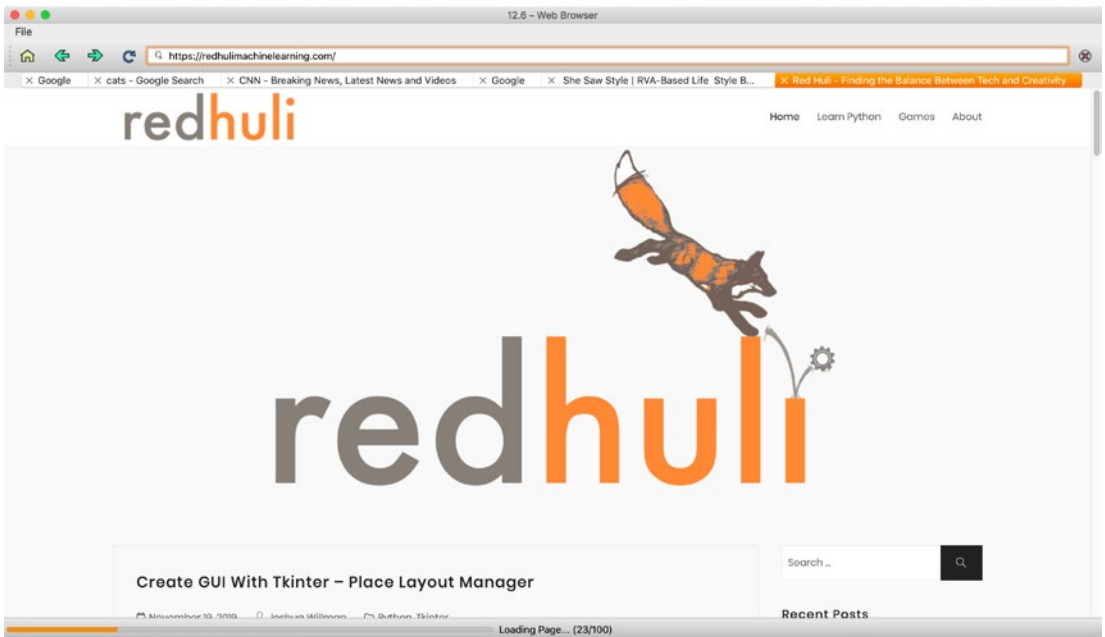
Figure 12-7. *The web browser GUI displaying the menubar, toolbar, different tabs, the logo for my blog, RedHuli, and the progress bar at the bottom*

Explanation

Two new classes are introduced in this application – **QUrl** is used for managing and constructing URLs, and **QWebEngineView** is used for creating the main component for rendering content from the Web, the web engine view (denoted as web_view in the code).

Before calling initializeUI(), we need to instantiate a few lists that will contain the new windows, web pages viewed, and URLs for each tab. This project also calls setWindowIcon() to include an application icon, but it will not be displayed on MacOS due to system guidelines.

There are three main methods that are called in `initializeUI()`. The first one is `createMenu()` for setting up the main menu and the status bar. The menu includes actions and shortcuts for creating new windows, new tabs, and closing the application.

Next is the `createToolbar()` method that creates the navigation bar of the web browser. The `tool_bar` includes buttons for navigating between web pages and a `QLineEdit` widget for entering and displaying URLs. Each button emits a signal when `triggered` that is connected to an appropriate slot. For example, if the `back_page_button` is pressed, the `backPageButton()` slot will be called.

```
def backPageButton(self):
    tab_index = self.tab_bar.currentIndex()
    self.list_of_web_pages[tab_index].back()
```

The current index of the tab we are viewing is stored in `tab_index`. The `back()` method is then called on the `web_view` object for that current tab. If the `tab_index` is not 0, then the user can navigate back through previously viewed web pages. The `back()` method is but one of several functions included in the QWebEngineView class. Other methods for navigation include `forward()`, `reload()`, and `stop()`, and these are also utilized for the other `tool_bar` buttons.

When the user enters a web address in the `QLineEdit` widget and presses the return key, we check to see if the URL begins with the correct scheme (such as `http`, `https`, or `file`). If a valid scheme is not present, `http` is appended to the beginning of the URL. If the URL conforms to standard encoding rules, a request is then sent to `load()` the web site.

Creating Tabs for the Web Browser

The third method, `createTabs()`, is used to handle creating the tab widget and the web view objects. First, we need to create the `QTabWidget` that will display each individual tab's web view. Refer back to Chapter 6 for more details on setting up tab widgets.

A few of the `tab_bar` widget's parameters are changed so that each tab includes a close button, and if only one tab remains, then the tab bar will not be displayed. This helps to make sure that there is always at least one tab in the main window. If a tab is closed, the `closeTab()` slot is called. The corresponding URLs and web view items for that tab are also removed from the `list_of_urls` and `list_of_web_pages` lists.

The first tab, `main_tab`, is created, added to the `tab_bar`, and then passed to the `setupTab()` method. The `tab_bar` widget is set as the central widget for the main window. When setting up a tab's page, we first need to create a web view object.

Creating the Web View

The `setupWebView()` function creates an instance of the `QWebEngineView` class, `web_view`, and sets the web view's URL to display the Google web page.

```
web_view.setUrl(QUrl("https://google.com"))
```

To create a basic instance of a web view in an application, you only need to create a `QWebEngineView` object, use the `load()` method to load the web page onto the web view widget, and then call show(). The following code shows the process for setting up a simple web view widget:

```
web_view = QWebEngineView()
web_view.load(QUrl("https://google.com"))
web_view.show()
```

Once the web page has loaded, the `urlChanged()` signal connected to `updateUrl()` changes the URL displayed in `address_line`. We can use the `loadFinished()` signal to tell the current tab to update its title using the `updateTabTitle()` slot and return the `web_view` widget.

Next, create the layout to hold the web view widget, append the current tab's URL and `web_page` object to the `list_of_urls` and `list_of_web_pages` lists, and set the layout for the current tab's page. The `web_page` object is the `web_view` widget that is returned from `setupWebView()` and displayed in the page of the tab.

Finally, to handle when a user switches between tabs, QTabWidget has the `currentChanged()` signal. If a different tab is selected, the connected slot, `updateUrl()`, will change the displayed URL in `address_line`.

Adding a QProgressBar to the Status Bar

In `setupWebView()`, a progress bar and label are also created that will be used to display the loading progress of a web page in the browser's status bar. When the `loadProgress()` signal is generated, the `updateProgressBar()` slot is called.

```
web_view.loadProgress.connect(self.updateProgressBar)
```

The loadProgress() function returns an int value that we can use to track how much of the page has loaded. While progress is less than 100, the progress bar and the label are both displayed and their values are set. The code for displaying the progress bar is shown as follows:

```
self.page_load_pb.setVisible(progress)
self.page_load_pb.setValue(progress)
```

The widgets are then added to the status bar.

```
self.status_bar.addWidget(self.page_load_pb)
```

When a page is finished loading, we call removeWidget() to remove the progress bar and the label. An example of the progress bar can be seen at the bottom of Figure 12-7.

Note   Creating a web browser is a very extensive task. There are many topics that are not included in this project, such as accessing HTTP cookies with QtWebEngineCore, working with the browser history with QWebEngineHistory, managing connections and client certificates, proxy support with QNetworkProxy, working with JavaScript, downloading content from web sites, and others. You are definitely encouraged to research these topics if you need to use QtWebEngine for more advanced projects.

Summary

In this chapter, you saw different GUI applications that build the structure for larger projects, such as the camera GUI or the web browser GUI. Other projects introduced components that you may be able to include in other programs, such as the directory viewer GUI, the clock GUI, or the calendar GUI. In the case of the Hangman GUI, a complete program was created so that it can hopefully give you ideas for other programs you may want to design.

We have explored a variety of topics for designing graphical user interfaces using PyQt5 and Python throughout this book – different types of widgets, classes, and layouts. We saw how to stylize your interfaces, how to add menus, and how to make creating an application simpler with Qt Designer.

We covered a few advanced topics such as working with the clipboard, SQL, and multithreaded applications, as well.

Appendix A will fill in more details about the PyQt5 classes used in this book, as well as a few other classes that there was no room to include in the previous chapters.

Appendix B is used to refresh your knowledge on the key Python concepts used in this book.

Your feedback and questions are always welcome. Thank you so much for traveling along with me on this journey to create this guide for you.

Reference Guide for PyQt5

PyQt is a Python binding for the Qt application framework maintained by Riverbank Computing Limited. A **binding** is an application programming interface (API) that provides the code to allow a programming language to use other libraries not native to that language. Qt is a set of C++ libraries and development tools, providing access to networking, threads, SQL databases, OpenGL and other graphics tools, XML, GUI development, and other features. This chapter focuses only on PyQt5, because as of this writing, PyQt4 is no longer supported.

Appendix A contains a reference for some of the tools, modules, and classes learned throughout this book, including

- Installing PyQt5 and Qt Designer
- A review for PyQt modules and classes
- An overview of Qt Style Sheets

More information about Riverbank Computing Limited and PyQt5 can be found at the following link:

https://riverbankcomputing.com/software/pyqt/intro

Installing PyQt5 and Qt Designer

Before beginning, make sure Python 3 is already installed on your system. If you have not already installed Python, or are not sure if you already have it installed, please check out Appendix B.

Also, the SIP binding generator is a tool used for creating bindings that allow Python to access the C++ classes. If you choose to download PyQt5 from the Python Package Index (PyPI) repository, then the `sip` module will automatically be downloaded, as well.

The following subsections focus on downloading PyQt5 from PyPI using `pip3`, which only operates on Python 3 environments.

Getting PyQt for Windows

To install PyQt5, enter the following command into the Command terminal:

```
pip3 install pyqt5
```

You can check to make sure PyQt downloaded properly by opening up the Python 3 interpreter and entering the following command:

```
>>> import PyQt5.QtWidgets
```

If no errors are returned, then the next step is to install Qt Designer. To get Qt Designer, enter the following line into the command line:

```
pip3 install pyqt5-tools
```

Next, we need to locate the Qt Designer executable. You will need to look for the site-packages folder inside the main Python directory. The path should look something similar to the following path:

```
C:\Users\your_info\Local\Programs\Python3_directory\Lib\site-packages\
pyqt5_tools
```

Once you have located the executable file labeled `designer`, create a shortcut so you will be able to easily access it for next time.

Getting PyQt for MacOS

Make sure that you already have Xcode downloaded. If not, you can get it from the App Store. Similar to Windows, to download PyQt5, enter the following command into the Terminal:

```
pip3 install pyqt5
```

If you are using Homebrew, then you can use the following line instead:

```
brew install pyqt5
```

Next, check to make sure PyQt downloaded properly by opening up the Python 3 interpreter and entering the following command:

```
>>> import PyQt5.QtWidgets
```

If no errors appear, then the next step is to install Qt Designer. For MacOS, the process to download Qt Designer is not as simple. There is no `pyqt5-tools` wheel compatible with MacOS. Therefore, your options are either to download Qt from `www.qt.io/download` or download a stand-alone version that is thankfully provided by Michael Herrmann at

`https://build-system.fman.io/qt-designer-download`

If you downloaded the stand-alone version, once the file finishes downloading, add the Qt Designer file to your Applications folder and open it up to get started.

Getting PyQt for Linux (Ubuntu)

For Ubuntu, enter the following command into the shell:

```
sudo apt-get install python-pyqt5
```

Open the Python 3 interpreter and enter the following statement to make sure that PyQt5 is installed properly:

```
>>> import PyQt5.QtWidgets
```

If no errors are returned, then we can install Qt Designer:

```
sudo apt-get install qttools5-dev-tools
```

Next, you need to locate Qt Designer, `designer`, in the following path:

```
/usr/lib/x86_64-linux-gnu/qt5/bin/
```

Finally, make a shortcut for Qt Designer so that you will be able to easily locate the application next time.

Other Methods for Getting PyQt

There are several other ways to install PyQt. One option is to build and install from source. If for some reason you must build from source, check out the following link for the PyQt reference guide:

`www.riverbankcomputing.com/static/Docs/PyQt5/`

Another method is to download PyQt through the Anaconda distribution. To install the package on any system, run:

```
conda install -c anaconda pyqt
```

Another option is to install PyQt5 in a virtual environment, but that won't be covered here.

Selected PyQt5 Modules

PyQt provides a range of modules that give you access to a wide array of tools, including basic GUI design, 2D and 3D rendering, multimedia content, networking, global positioning, and more. For basic GUI development, you will primarily use the QtWidgets, QtGui, and QtCore modules. Table A-1 lists the modules covered throughout the book, as well as a few extra you should check out.

For a full list of PyQt5's top-level modules, check out the following link:

www.riverbankcomputing.com/static/Docs/PyQt5/module_index.html

Table A-1. *Table of select PyQt modules*

Module Name	Description
QtWidgets	Provides the widgets and other classes for creating desktop-style UIs.
QtCore	Contains a variety of extra classes, including the essential non-GUI classes, such as ones for Qt's signal and slot system.
QtGui	Contains classes for 2D graphics and imaging, event handling, and window system integration.
QtPrintSupport	Provides cross-platform support for configuring and connecting to printers.
QtNetwork	Provides classes for writing communications protocols using UDP or TCP.
QtMultimedia	Contains the classes for multimedia content, cameras, and radios.
QtMultimediaWidgets	Provides additional classes that increase the functionality of multimedia-related widgets.
QtWebEngineCore	Contains the core classes used by other WebEngine modules.

(*continued*)

Table A-1. (*continued*)

Module Name	Description
QtWebEngineWidgets	Classes that can be used to create a Chromium-based web browser.
QtSql	Provides classes for working with SQL databases.
sip	Tools used for creating Python bindings for C ++ libraries (which is the language Qt is written in).
uic	Contains classes used for handling the .ui files created by Qt Designer.

Selected PyQt Classes

There are hundreds of PyQt classes. The following section lists the key classes and widgets that can be found throughout this book. Each subsection either lists tables with commonly used methods and signals, or a link to where you can find more information about the class.

For a list of all the PyQt classes, check out the following link:

www.riverbankcomputing.com/static/Docs/PyQt5/sip-classes.html

Although it is written for C++, the Qt classes' documentation is generally more detailed. If you want more information about Qt classes, you can also check out

https://doc.qt.io/qt-5/classes.html

Classes for Building a GUI Window

With PyQt, you can create a new class that inherits from any of the widget classes. However, for a general GUI application, you will need to create only one instance of QApplication, and create a class that inherits from either QWidget, QMainWindow, or QDialog to create the application's window.

QApplication

QApplication is responsible for handling the initialization and finalization of widgets in graphical user interfaces. If you are making QWidget-based applications, then you will need to create an instance of QApplication before creating any other objects related to the GUI.

Some of QApplication's responsibilities include initializing an application to conform to a user's desktop settings, event handling, defining the GUI's style, working with the clipboard, and keeping track of all the application's windows.

If you are creating an application that does not need a GUI and can be run through the command line, then you should consider using **QCoreApplication**, instead.

QWidget

The QWidget class is the base class for all of PyQt's graphical user interface's objects. A widget created from the QWidget class is able to receive input from the mouse, keyboard, and other events and able to paint itself on the screen. Widgets that are not embedded in a parent widget are considered to be a window complete with a title bar and a frame. The QWidget class is a subclass of QObject and QPaintDevice. Some of QWidget's more commonly used methods can be found in Table A-2.

Table A-2. *Selected methods from QWidget*

Method	Description
addAction(action)	Add an action to the widget.
close()	Close the widget.
height()	Holds the widget's height.
width()	Holds the widget's width.
move(x, y)	Sets the location of the widget to (x, y).
rect()	Holds the geometry of the widget minus the frame.
setDisabled(bool)	If True, the widget is disabled.
setEnabled(bool)	If True, the widget is enabled.
setFont(font)	Sets the font of the widget's text.
setLayout(layout)	Sets the layout manager for the widget.
setGeometry(x, y, width, height)	Sets the widget's location, (x, y), and its size, width and height.
setStyleSheet(styleSheet)	Sets the styleSheet for the widget.
setToolTip(text)	Sets the widget's tool tip.
repaint()	Repaints the widget immediately by calling paintEvent().
showFullScreen()	Displays the widget in full screen mode.
update()	Updates the widget.

Event Handling

Events are typically caused by users. These can include moving a mouse, pressing a key, or resizing the window. The widgets in an application need to respond to the event caused by the user's actions. The events are typically already handled, but you sometimes may find yourself needing to reimplement event handlers to supply further behavior or content for the widgets. Table A-3 lists a few commonly used event handlers.

Table A-3. *Some event handlers used for supplying behavior to QWidget objects*

Event Handler	Description
paintEvent()	Called whenever a widget needs to be repainted.
resizeEvent()	Called when a widget has been resized.
mousePressEvent()	Called when a mouse button is pressed while the mouse cursor is inside of the widget. Which mouse button is clicked can be specified in the event.
mouseReleaseEvent()	Called when a mouse button is released. A widget that receives this event is dependent on receiving the mouse press event.
mouseDoubleClickEvent()	Called when a widget is double-clicked.
mouseMoveEvent()	Called when the mouse moves while the button is held down. If setMouseTracking() is True, events are sent even when no buttons are pressed.
enterEvent()	Called when the mouse enters a widget's space.
leaveEvent()	Called when the mouse leaves a widget's space.
keyPressEvent()	Called when a key is pressed.
keyReleaseEvent()	Called when a key is released.
focusInEvent()	Called when a widget gets the keyboard focus.
focusOutEvent()	Called when a widget loses the keyboard focus.
closeEvent()	Called when either a widget or the window is closed.

QMainWindow

The QMainWindow class provides the framework for building an application, complete
with functions for adding a menubar, toolbars, a status bar, and dock widgets. Menu
and toolbar items are created using QAction. QMainWindow already has its own layout,
to which you must add a central widget as the center area of the application's window.
Some of QMainWindow's methods can be seen in Table A-4.

Table A-4. *Select methods from QMainWindow*

Method	Description
addDockWidget(area, dockwidget)	Creates a dockwidget in the main window in the specified area.
addToolBar(toolbar)	Creates a toolbar for the main window. An area can also be specified.
menuBar()	Returns the main window's menubar.
setStatusBar(statusbar)	Creates the statusbar for the main window.
setCentralWidget(widget)	Sets the window's central widget.
setWindowIcon(icon)	Sets the window's icon.
setWindowTitle(text)	Sets the window's title.

QDialog

Dialog boxes provide a top-level window generally to obtain feedback quickly from a
user. QDialogs can be modal or modeless. Modal dialogs are often used when selecting
an option in the dialog that will return a value. That value could then be used to save a
file, close a document, or cancel an action. Table A-5 lists the QDialog methods useful
for creating modal or modeless dialog boxes.

QDialog is the base class for dialog boxes, including QColorDialog, QFileDialog,
QFontDialog, QInputDialog, QMessageBox, QProgressDialog, and QErrorMessage.

Table A-5. *Select methods for QDialog*

Method	Description
accept()	Hides the modal dialog and returns True.
reject()	Hides the modal dialog and returns False.
exec_()	The dialog is shown as a modal dialog and blocks the user from any further action until the dialog is closed.
show()	The dialog is a modeless dialog, returning control to the user immediately.

QPainter

The QPainter class is responsible for handling drawing in PyQt, being able to draw simple lines and complex shapes onto widgets and other paint devices. QPainter is most commonly used in a QWidget's paintEvent() and for handling pixmaps and images. Table A-6 displays some of QPainter's methods for drawing.

Table A-6. *Methods selected from QPainter*

Method	Description
begin(device)	Begins painting on the paint device.
end()	Ends painting. Resources used while painting are released.
save()	Saves the current painter state. save() must be followed be restore(), which returns the current painter state.
drawArc(QRectF, startAngle, spanAngle)	Draws an arc defined by the QRectF rectangle, startAngle, and spanAngle.
drawChord(QRectF, startAngle, spanAngle)	Draws a chord defined by the QRectF rectangle, startAngle, and spanAngle.
drawEllipse(QPointF, x_rad, y_rad)	Draws an ellipse at QPointF center, with radius x_rad and y_rad.
drawLine(x1, y1, x2, y2)	Draws a line from point (x1, y1) to (x2, y2).
drawPath(path)	Draws a path specified by QPainterPath path.

(*continued*)

Table A-6. (*continued*)

Method	Description
drawPie(QRectF, startAngle, spanAngle)	Draws a pie defined by the QRectF rectangle, startAngle, and spanAngle.
drawPixmap(x, y, pixmap)	Draws a pixmap at (x, y).
drawPoint(x, y)	Draws a point at (x, y).
drawRect(x, y, width, height)	Draws a rectangle at (x, y) with width and height.
drawRoundedRect(QRectF, x_rad, y_rad)	Draws a rectangle with rounded corners specified by QRectF, with radius x_rad and y_rad.
drawText(QPointF, text)	Draws text at QPointF point.
fillRect(QRectF, brush)	Fills in a QRectF rectangle with the brush color.
rotate(angle)	Rotates the coordinate system clockwise by angle (in degrees).
setBrush(brush)	Sets the painter's brush.
setPen(pen)	Sets the painter's pen.
setFont()	Sets the painter's font.

Layout Managers

Using PyQt's layout managers makes the process of arranging widgets much easier, compared to manually specifying each widget's size, position, or resizeEvent() event handler. Using layout managers is generally a good start for positioning widgets, although you may still need to adjust a widget's size policy, or add stretching or spacing to a layout.

The following classes inherit from the **QLayout** class, which is the base class for layout managers:

- QBoxLayout – Arranges child widgets into a row (horizontally), or into a column (vertically)

    a.   QHBoxLayout – Arranges widgets horizontally

    b.   QVBoxLayout – Arranges widgets vertically

- QGridLayout – Orders widgets in a grid of rows and columns

- QFormLayout – Lays out widgets into a form-like structure with labels
  and their associated input widgets

There is also **QStackedLayout** which was not covered in this book. The convenience QStackedWidget class is built on top of the QStackedLayout. Table A-7 lists commonly used methods from the layout classes.

Table A-7. *Selected methods for the different layout managers*

Method	Class	Description
addWidget(widget, stretch, alignment)	QBoxLayout	Add widget to the end of the layout with stretch factor and alignment.
addWidget(widget, row, column, rowSpan, columnSpan alignment)	QGridLayout	Add widget at row, column with (optional) rowSpan and columnSpan and alignment.
addRow(label, field)	QFormLayout	Add a new row with given label and field (input widget).
addLayout(layout, stretch)	QBoxLayout	Adds a layout to the end of the box. Creates a nested layout.
addLayout(layout, row, column, alignment)	QGridLayout	Adds a layout at position (row, column). Creates a nested layout.
addSpacing(int)	QGridLayout, QBoxLayout	Adds a nonstretchable area (a QSpacerItem) of int value to the layout.
addStretch(int)	QBoxLayout	Adds a stretchable area (a QSpacerItem) of int value to the layout.
setSpacing(int)	QLayout	Sets the space between widgets in the layout. Inherited from QLayout.
setContentMargins(left, top, right, bottom)	QLayout	Sets the left, top, right, and bottom margins around the layout.

Button Widgets

Buttons are one of the main tools used in a GUI for interaction, giving an application feedback about a user's decisions. Buttons in PyQt can display text or icons and are checkable. The following classes inherit from the base class for button widgets, **QAbstractButton**:

- `QPushButton` – A command button used to tell the computer to perform some action

- `QCheckBox` – Provides an option button that is checkable, and generally used for enabling/disabling features in an application

- `QRadioButton` – Similar to checkboxes, but are mutually exclusive

- `QToolButton` – Typically used in a toolbar, tool buttons provide quick-access buttons for selecting commands or options

For managing and organizing multiple buttons, the QButtonGroup class can act as a container for creating exclusive buttons (the default setting). Table A-8 lists some of the more commonly used methods for button widgets.

Table A-8. *Selected methods for the different button widgets*

Method	Description
`setIcon(icon)`	Sets the widget's `icon`.
`setText(text)`	Set's the widget's `text`.
`setAutoExclusive(bool)`	Enables auto-exclusivity for buttons in a group.
`setCheckable(bool)`	Sets whether the button is a toggle button or not.
`setChecked(bool)`	Sets whether the button is checked or not.
`isChecked()`	Indicates whether the button is checked or not (is `setCheckable()` is True).
`text()`	Holds the buttons text.

The signals of the button widgets are listed in Table A-9.

Table A-9. *Signals for the different button widgets*

Signal	Class	Description
clicked(bool)	QAbstractButton	Signal emitted when the button is pressed and released.
pressed()	QAbstractButton	Emitted when the left mouse button clicks the button.
released()	QAbstractButton	Signal emitted when the left mouse button is released.
toggled(bool)	QAbstractButton	Emitted when a checkable button changes its state.
stateChanged(bool)	QCheckBox	Emitted when the checkbox's state changes.
triggered(action)	QToolButton	Signal emitted when the action is triggered.

Input Widgets

There are quite a few widgets that are provided by PyQt for getting input from the user. These widgets provide different means for gathering information, such as text entry, or selecting values with sliders, combo boxes, and spin boxes.

Combo Boxes

The QComboBox class presents a user with a list of selectable options in a compact, drop-down menu. When the combo box is not being interacted with, all items except for the current item selected are hidden from view. Some common methods for QComboBox can be found in Table A-10. The **QFontComboBox** widget is another type of combo box that inherits QComboBox and is used for selecting a font family.

Table A-10. *Select methods from the QComboBox class*

Method	Description
addItem(text)	Appends an item to the list with text.
addItems(list(text))	Appends a list of items to the combo box.
currentIndex()	Holds the index of the currently selected item.
currentText()	Holds the text of the currently selected item.
insertItem(index, text)	Inserts the text into the combo box at the given index.
setItemText(index, text)	Sets the text for the item at the given index.
removeItem(index)	Removes the item at the given index.
clear()	Clears all items from the combo box.
setEditable(bool)	If True, the contents of the combo box are editable.

Table A-11 displays select signals for the combo box classes.

Table A-11. *Commonly used signals from the QComboBox and QFontComboBox classes*

Signal	Description
currentIndexChanged(index)	Emitted if the current item in the combo box has changed.
currentTextChanged(text)	Signal emitted if the current item in the combo box has changed. Returns text.
activated(index)	Emitted only if the user interacts with an item.
highlighted(index)	Emitted when an item in the combo box is highlighted.
textActivated(text)	Signal emitted when the user chooses an item.
currentFontChanged(font)	Emitted when the current font changes.

QLineEdit

The QLineEdit widget provides a single line for entering and editing plain text. Although not listed in the following tables, QLineEdit comes with clear(), selectAll(), cut(), copy(), paste(), undo(), and redo() slots already built-in. Table A-12 displays a few the QLineEdit class's methods.

Table A-12. *Methods from the QLineEdit class*

Method	Description
text()	Retrieves the current text in the line edit.
setAlignment(alignment)	Sets the alignment of the text displayed in the widget.
setPlaceholderText(text)	Displays placeholder text while line edit is empty.
setEchoMode(mode)	mode describes how the contents of a line should be displayed. Set mode to QLineEdit.Password to mask characters.
setMaxLength(int)	Sets the maximum length of characters.
setTextMargins(left, top, right, bottom)	Sets the text margins for the text displayed in the line edit.
setDragEnabled(bool)	If True, dragging selected text in the line edit is permitted.

A few common signals for QLineEdit can be seen in Table A-13.

Table A-13. *Commonly used signals from the QLineEdit class*

Signal	Description
returnPressed()	Emitted when the Enter key is pressed. If a validator is set, then a signal is only emitted if the text is accepted.
textChanged(text)	Signal is emitted when the text changes.

Text Editing Widgets

The two text editing classes, QTextEdit and **QPlainTextEdit**, provide tools and functionality for displaying and editing larger bodies of text. QTextEdit also has the added benefit of being able to work with rich text, graphics, and tables. Select methods for the two classes are displayed in Table A-14. Both classes are similar to QLineEdit, because they already have editing features built-in.

Also worth noting is the **QTextBrowser** class, which inherits QTextEdit. QTextBrowser only allows read-only mode, but includes hypertext navigation functionality so that users can click links and follow them.

Table A-14. *Select methods from QTextEdit and QPlainTextEdit*

Method	Description
find(text, flags)	Finds the next occurrence of text in the text edit.
print(printer)	Print the text edit's document to the printer.
setPlaceHolderText(text)	Sets placeholder text for text edit.
setReadOnly(bool)	If True, the text edit is set to read-only.
toPlainText()	Returns the text of the text edit as plain text.
zoomIn(range)	Zooms in on the text.
zoomOut(range)	Zooms out on the text.

Commonly used signals for the text editing widgets can be found in Table A-15.

Table A-15. *Select signals from QTextEdit and QPlainTextEdit*

Signal	Description
selectionChanged()	Signal emitted when the text selected in the text edit changes.
textChanged()	Emitted whenever the contents of the text edit change.

Spin Box Widgets

Spin boxes allow users to choose values within a given range by clicking up/down
buttons to cycle through the widget's values. Users can also manually type in values into
the provided line edit. The **QAbstractSpinBox** class is the base class for the following
classes:

1. QSpinBox – Handles integers.

2. QDoubleSpinBox – Similar to QSpinBox but is used for floating-
   point values.

3. QDateTimeEdit – A spin box widget for selecting dates and times.
   Use setDisplayFormat() to set the format used for displaying the
   dates and time.

4. `QDateEdit` – A spin box that displays only dates. Inherits from `QDateTimeEdit`.

5. `QTimeEdit` – A spin box that displays only times. Inherits from `QDateTimeEdit`.

Some of the methods for the QSpinBox and QDoubleSpinBox classes are listed in Table A-16. The QDateTimeEdit and other spin box widgets have similar methods.

Table A-16. *Select signals from QSpinBox and QDoubleSpinBox. The value* `val` *refers to integers for QSpinBox and floating-point numbers for QDoubleSpinBox*

Method	Description
`setValue(val)`	Sets the value `val` of the spin box.
`setMinimum(val)`	Sets the minimum value `val` of the spin box.
`setMaximum(val)`	Sets the maximum value `val` of the spin box.
`setPrefix(str)`	Adds a prefix to the start of the displayed value.
`setSuffix(str)`	Adds a suffix to the end of the displayed value.
`setRange(min, max)`	Sets the minimum and maximum range values.
`setSingleStep(val)`	The spin box's value is incremented/decremented by `val` when the arrow keys are pressed.

The QSpinBox and QDoubleSpinBox signals are found in Table A-17.

Table A-17. *Signals from QSpinBox and QDoubleSpinBox*

Signal	Description
`valueChanged(val)`	Signal emitted when the value changes. Provides the new value's `val`.
`textChanged(text)`	Signal emitted when the value changes. Provides the new value's `text`.

Slider Widgets

The following widgets are different in appearance, but are actually quite similar in function. Widgets that inherit from the **QAbstractSlider** class are used for selecting integer values within a bounded range. Classes that inherit QAbstractSlider include the following:

1. `QDial` – Provides a rounded range controller for selecting or adjusting values. An example of QDial can be seen in Figure A-1.

2. `QScrollBar` – Provides horizontal or vertical scrollbars that the user can use to access other parts of a document that are wider than the widget used to display it.

3. `QSlider` – Creates the classic horizontal and vertical slider widgets for controlling values within a specified range.

Table A-18 shows some of the methods of the QAbstractSlider base class.

Table A-18. *Select methods from QAbstractSlider*

Method	Description
value()	Holds the sliders current value.
setMinimum(int)	Sets the minimum value of the slider.
setMaximum(int)	Sets the maximum value of the slider.
setOrientation(orientation)	Sets the orientation, Qt.Horizontal or Qt.Vertical.
setSingleStep(int)	The slider's value is incremented/decremented by int when the arrow keys are pressed.
setTracking(bool)	If True, the slider's position can be tracked.
setSliderPosition(int)	Sets the current position of the slider.
setValue(int)	Sets the current position of the slider to int. If tracking is enabled, then this has the same value().

Signals of the QAbstractSlider class can be found in Table A-19.

Table A-19. *Signals from QAbstractSlider*

Signal	Description
valueChanged(val)	Signal emitted when the value changes. Provides the new value's val.
rangeChanged (min, max)	Signal emitted when the range has changed with new minimum and maximum values.
sliderMoved(val)	Emitted when the slider is pressed down and the slider moves.
sliderPressed()	Emitted when the slider is pressed down.
sliderReleased()	Emitted when the slider is released.

Display Widgets

The following widgets are all used for different purposes, but each has one major characteristic in common – they are all used for displaying information to the user.

QLabel

The QLabel is a versatile widget. Although a label provides no user interaction functionality, QLabel is able to display plain or rich text, pixmaps, and even movies. Labels provide a number of methods for configuring their appearance. Table A-20 lists a few of those methods.

Table A-20. *Select methods from QLabel*

Method	Description
setPicture(picture)	Sets the label content to picture.
setPixmap(pixmap)	Sets the label content to pixmap.
setMovie(movie)	Sets the label content to movie.
setText(text)	Sets the label content to text.
setAlignment(alignment)	Sets the alignment of the label's content.
setIndent(int)	Sets the number of pixels that the label's text is indented.
setMargin(int)	Sets the label's margins.

QProgressBar

Progress bars are used to give visual feedback to the user about the progress of a computer operation. Progress bars can be displayed vertically or horizontally. Table A-21 shows some of the QProgressBar class's methods.

Table A-21. *Select methods for the QProgressBar class*

Method	Description
value()	Holds the progress bar's current value.
setMinimum(int)	Sets the progress bar's minimum value.
setMaximum(int)	Sets the progress bar's maximum value.
setRange(min, max)	Sets the minimum and maximum values.
setOrientation(orientation)	Sets the orientation, Qt.Horizontal or Qt.Vertical.
setTextVisible(bool)	If True, the current completed percentage is displayed.

QProgressBar has one signal, valueChanged(int), that is emitted when the value shown in the progress bar changes.

QGraphicsView

The QGraphicsView class provides a widget for displaying the contents of a QGraphicsScene. As the one part of Qt's Graphics View Framework, the QGraphicsView's responsibility is to display the items of a graphics scene in a scrollable window. The QGraphicsScene's duty is to manage the items in a scene. QGraphicsItem (or one of its subclasses) provides the items for a scene.

If you are interested in learning more about the Graphics View Framework, check out the following link:

https://doc.qt.io/qt-5/graphicsview.html

QLCDNumber

The **QLCDNumber** widget displays numbers in a seven-segment LCD display. An example of this can be seen in Figure A-1. The display can visualize decimal, hexadecimal, octal, and binary numbers. The LCD display can only display certain characters, so if an illegal character is passed, a space will be displayed in place of the character.

Figure A-1. *Example of the QLCDNumber and QDial widgets*

Table A-22 lists a few of QLCDNumber's methods.

Table A-22. *Select methods from the QLCDNumber class*

Method	Description
value()	Retrieves the LCD's displayed value.
intValue()	Retrieves the displayed value rounded to the nearest integer value.
display(val)	Displays the value val in the display. val can be floating-point, integer, or string type.
setMode(mode)	Sets the mode of the LCD to display Bin, Oct, Dec, or Hex values.
setSmallDecimal Point(bool)	If True, the decimal is drawn between two digits.

QLCDNumber has the overflow() signal, which is emitted when the widget is asked to display a number or string that is too long.

Item Views

The following model view classes provide the means to display items in lists, tables, or tree structures. They must be used alongside a model class as part of Qt's model/view framework.

1. QListView – Provides a list and icon view for displaying items from a model

2. QTableView – Provides a table for displaying items from a model

3. QTreeView – Provides a hierarchical tree architecture for displaying items from a model

These classes all inherit from the **QAbstractItemView** class. Using signals and slots, item views created from QAbstractItemView are able to interact with models that use **QAbstractItemModel**. Each of the item views has their own methods for working with rows, columns, headers, and items. Views use the indices to manage items. You can find some of QAbstractItemView's methods listed in Table A-23.

PyQt also offers convenience item-based classes for each of the different types of views – QListWidget, QTableWidget, and QTreeWidget. Items are added to the widgets by using QListWidgetItem, QTableWidgetItem, or QTreeWidgetItem.

Table A-23. *Select methods for the QAbstractItemView base class*

Method	Description
clearSelection()	All items selected are deselected.
selectAll()	Selects all the items in the view.
setCurrentIndex(index)	Sets the item at index as the current item.
update(index)	Updates the area at the given index.
setAlternatingRowColors (bool)	If True, the background is drawn with alternating colors.
setAcceptDrops(bool)	If True, items can be dropped into the view.
setDragEnabled(bool)	If True, items can dragged around in the view.
setIconSize(size)	Sets the size of icons.
setItemDelegate(delegate)	Sets an item delegate for the view's model/view framework.
setModel(model)	Sets the model for the view.

Select signals for QAbstractItemView can be found in Table A-24.

Table A-24. *Select methods for the QAbstractItemView base class*

Signal	Description
activated(index)	Signal emitted when the item at index is activated by the user.
clicked(index)	Emitted when the left mouse button is clicked on an item in the view (specified by index).
doubleClicked(index)	Emitted when a mouse button is double-clicked on an item in the view (specified by index).
entered(index)	Signal emitted when the mouse cursor enters the item at index. Turn on mouse tracking to use.
pressed(index)	Signal emitted when a mouse button is pressed on an item at index.

Container Widgets

PyQt provides a few container widgets for maintaining control over groups of widgets. Containers can be used to manage input widgets and make the process of organizing a group of widgets simpler, or simply as a decorative widget for separating groups of widgets. Once a container is created, a layout manager still needs to be applied to the container widget itself.

Containers with Frames

QFrame widgets can enclose and group widgets, as well as function as placeholders in windows. Using frames, you can set the appearance of other widgets to have raised, sunken, or flat appearances. The QFrame class is used as the base class for a few other container classes, including **QToolBox** and **QStackedWidget**. Table A-25 lists a few of QFrame's methods.

Table A-25. *Select methods for QFrame*

Method	Description
setFrameRect(QRect)	Sets the rectangle that the frame is drawn in.
setFrameShadow(shadow)	Sets the frame's shadow, such as Plain, Raised, or Sunken.
setFrameShape(shape)	Sets the frame's shape, such as Box, Panel, HLine, or VLine.
setLineWidth(int)	Sets the width of line drawn around the frame.

QToolBox widgets provide a series of pages or compartments in a column. To navigate through each of the pages, a tab is included at the top of each page. By clicking the next tab, the user can view a new tab's contents.

Table A-26. *A few of the QToolBox class's methods*

Method	Description
addItem(widget, text)	Adds the widget in a new tab at the bottom of the toolbox.
insertItem(index, widget, text)	Inserts the widget in a new tab at the given index.
indexOf(widget)	Returns the index of the specified widget.
setCurrentIndex(index)	Sets the index to a new item's index.
setCurrentWidget(widget)	Makes the widget the current widget displayed in the toolbox.

When the item in a QToolBox is changed, the currentChanged(index) signal is emitted.

The QStackedWidget has a similar function to QToolBox, displaying multiple widgets stacked on top of one another to conserve space in a window. However, there is a key difference: QStackedWidget does not provide a means for the user to switch between tabs. Therefore, other widgets, such as a QComboBox or a QListWidget, are used to navigate through the different pages.

The QTabWidget is another container class that is similar to QStackedLayout, but provides the tabs necessary to switch pages.

Finally, QGroupBox widgets typically group together collections of radio buttons and checkboxes. The main visual difference from the QFrame class is the addition of a title.

QScrollArea

A scroll area can be added onto a child widget to display the contents within a frame. If the size of the frame changes, the scrollbars will appear, allowing the user to still view the entire child widget. The manner in how the scrollbars appear can be controlled with **QAbstractScrollArea**'s size policies. You can find a few of QScrollArea's methods in Table A-27.

Table A-27. *Select methods for QScrollArea*

Method	Description
ensureVisible(x, y, xmargin, ymargin)	Ensures the specified (x, y) coordinate with margins xmargin and ymargin remains visible in the viewport.
setAlignment(alignment)	Sets the alignment of the scroll area's widget.
setWidget(widget)	Sets the scroll area's widget.
setWidgetResizable(bool)	If False, the scroll area abides by the child widget's size.

QMdiArea

For multiple-windowed GUIs (MDIs), the QMdiArea provides the container for displaying multiple windows inside a single application window. Subwindows are instances of the QMdiSubWindow class and can be arranged in tiled or cascading patterns. The subwindows can work together, relaying information back and forth. A context menu could also be added to the MDI area widget as a means to conveniently switch between windows. Methods to help you get started using QMdiArea can be found in Table A-28.

Table A-28. *List of select QMdiArea methods*

Method	Description
addSubWindow(widget)	Adds widget as a new subwindow to the MDI area.
activeSubWindow()	Returns the active subwindow.
cascadeSubWindow()	Arranges subwindows in a cascade pattern.
tileSubWindows()	Arranges subwindows in a tiled pattern.
removeSubWindow(widget)	Removes widget from the MDI area, where widget is a subwindow.
setBackground(background)	Sets the QBrush background for the MDI area.
subWindowList(subwindows)	Returns a list of subwindows.
setTabsClosable(bool)	If True, close buttons are placed on each tab in the tabbed view.
setTabsMovable()	If True, tabs within the tabbed view are movable.

Qt Style Sheets

For a great reference to all of the widgets and properties that can be manipulated with Qt Style Sheets, have a look at the following link:

https://doc.qt.io/qt-5/stylesheet-reference.html

Style sheets allow for customizing many aspects and behaviors of widgets. Table A-29 lists many of the properties that can be modified. Widgets support only certain properties, so be sure to check out Qt's documentation if you are not sure about which properties you can change.

Table A-29. *List of properties that can be influenced using Qt Style Sheets*

Property	Description

alternate-background-color	The alternate background color for QAbstractItemView widgets.
	QListView{
	alternate-background-color: blue;
	background: grey
	}

background	Shorthand for setting the background.

background-color	Background color used for the widget.
	QPushButton{
	background-color: #49DE1F
	}

background-image	The background image used for the widget.
	QFrame{
	background-image: url(images/black_cat.png)
	}

border	Shorthand for setting the widget's border.
	QComboBox{
	border: 2px solid magenta
	}

(*continued*)

Table A-29. (*continued*)

Property	Description
`border-top, border-right,` `border-bottom, border-left`	Shorthand for specifying sides of the widget's border.
`border-color`	The color for all sides of the widget's border.
`border-image`	Specifies an image to fill the border.

`border-radius`	The radius of the border's corners.
	`QTextEdit{`
	`border-width: 1px;`
	`border-style: groove;`
	`border-radius: 3px`
	`}`

`border-style`	Specifies the style for all of the border's edges.
`border-width`	Specifies the width for all of the border's edges.
`color`	The color used for rendering text.

`font`	Shorthand for defining a widget's font.
	`QRadioButton{`
	`font: bold italic large "Helvetica"`
	`}`

`font-family, font-size,` `font-style, font-weight`	Other properties used to individually set a font's features.
`height, width`	The height and width of a widget.
`icon-size`	The width and height of a widget's icon.
`image`	The image drawn on a widget. Can use URL or SVG.
`left, right, top, bottom`	Moves a widget by a certain offset relative to the parent's edge.
`margin`	Specifies the widget's margins. Just like `border`, specific sides can also be set.
`max-height, max-width`	The widget's maximum height or width.

(*continued*)

Table A-29. (*continued*)

Property	Description
`min-height, min-width`	The widget's minimum height or width.
`outline`	The outline is used to draw a widget's border. Can also specify color, style, and radius.
`padding`	Specifies the widget's padding. Just like `border`, specific sides can also be set.
`selection-color`	The foreground color of selected items to text.
`spacing`	Sets the internal spacing in a widget.
`text-align`	Specifies the alignment of text and icons inside of a widget. `QPushButton{` `text-align: right` `}`

Summary

Throughout this book, you have seen many of PyQt's foundational classes for building graphical user interfaces. Appendix A serves as a reference to help you analyze the programs found in this book and to learn more about the widgets, layouts, and style sheets used to design and build each application. The classes and methods contained here act as a guide to get you thinking about ways to improve on the programs in this guide and to help you to make your very own applications.

There is simply not enough room to include every class, method, or signal in this guide. As you follow along with the examples, use this appendix as a resource to help you learn and find out more about the possibilities of PyQt. If the answer isn't provided for you here, follow the links, search on the Internet, or send me an e-mail.

Python Refresher

Python is a powerful and versatile open source programming language with a standard library packed with built-in modules, from providing access to system functionality such as file I/O to Internet data handling, to development tools, and much more. On top of the standard library, Python is also host to an extensive number of third-party modules which can be found in the Python Package Index (PyPI) repository. No matter what your project may be, you can almost be guaranteed to find a module to fit your needs. Of course, you could also create your own module and share it with the Python community.

Some of the uses for Python include

- Web and Internet development
- Database access
- GUI development
- Scientific and numeric applications
- Network programming
- Software and game development

The Python language also has many other noteworthy features, such as follows:

1. Python is an interpreted programming language. An **interpreted language** is one in which the code does not need to be compiled before it is run. The compiling step turns the code into machine language where the instructions can be directly executed by a computer's central processing unit (CPU). Since Python does not have a compilation step, the process to edit, test, and debug code is much faster.

2. Python is a language well-suited for creating dynamic applications. This is because Python is both a **dynamically typed** language – type checking occurs only as the code is being run and variables can change types over time – and allows for **dynamic binding** – where methods that are called on objects are only checked during runtime.

3. While it is an object-oriented programming language, it does also support other programming paradigms such as procedural programming.

Note   This section is not designed to act as a complete tutorial for Python. It focuses mainly on reviewing some of the basic data types, data structures, methods, and ideas that could be useful when creating your own GUI applications and reviewing other concepts that are used throughout this book.

First, let's get started by seeing if you have an updated version of Python on your computer. If not, take a moment to install it.

Installing Python

As of this writing, the current version of Python 3 is version 3.8, but the code in this book is also capable of being run on Python 3.6 or later.

Python is a cross-platform programming language and is therefore able to run on a number of different hardware platforms and operating systems. Whether you are using Windows, MacOS, or some form of Linux, the simplest way to download the latest version of Python is to go to https://python.org/downloads/ and find the installer for your platform.

Getting Python for Windows

To first check if Python is installed on your system, open up a Command window and enter the command python3 (all lowercase). If Python is already installed, you will get a notice telling you the current version that is on your system and a chevron, >>>, prompting you to enter a Python command. Otherwise, you will get an error message saying python3 is not a recognized command.

If Python is not already on your computer, then downloading Python for Windows is fairly easy. Select the installer right for your system from `python.org`, download it, and follow along with the instructions in the dialog that appears. Be sure to include Python in your PATH by checking the box like in Figure B-1.

Once installation is complete, try entering the `python` command into the Command window again to make sure Python 3 is installed properly.

Figure B-1. *On Windows, make sure to check the box "Add Python 3.x to PATH". This ensures that the interpreter will be placed in the correct directory*

Getting Python for MacOS

Python already comes installed on MacOS, but will probably need to be updated. To check out which version of Python 3 you already have installed on your computer, open up the Terminal application and enter `python3`. Similar to Windows, if you get an error message, Python 3 is not installed on your system.

Doing some research online to find out what method works best for you is a good idea. Otherwise, you could end up with multiple versions of the Python interpreter on your computer.

Downloading Python directly and installing the interpreter from https://python. org/ is one option. After the file downloads, simply run the installer. Once the installer is complete, check to make sure Python is installed by opening up the Terminal window and entering python3 on the command line.

Another option is to download the Homebrew package manager. A package manager can be useful for maintaining, updating, and organizing the different software on your computer. If you are interested in using a package management system, first open a Terminal window on your Mac. To get Homebrew, go to https://brew.sh and copy and paste the code into Terminal to install it. Once the installation is complete, enter the following code (without the $ symbol) into the terminal window to download Python using Homebrew:

```
$ brew install python3
```

Once the downloading process is complete, enter python3 into the Terminal window to launch the Python interpreter.

If you have multiple versions of Python on your computer, you can also use Homebrew and a tool called pyenv to manage them.

Getting Python for Linux

If you are using Linux, then you probably already have a version of Python installed on your system. The Linux platform has a number of different distributions – Ubuntu, Fedora, Debian, and a few others.

To check if Python is already installed and, if so, find out what the current version is, open the Terminal application on your system and enter python3. If Python 3 is installed, then your current version should be displayed along with the chevron prompt, >>>.

If you need to download or update your version of Python, you can download the most recent release from python.org. In Linux, it is also possible to download Python from the Terminal. However, the process for each distribution of Linux can be slightly different. There are a number of great tutorials online that you can follow along with if you choose to go this route to get Python installed on your computer.

Data Types in Python

There are several data types already built into Python, including numeric, string, and Boolean types. The following section reviews what these data types look like, as well as takes a look at a few of Python's built-in functions to work with them.

Numeric Data Types

When working with numbers, Python has **integer**, **floating-point,** and **complex number** data types (Table B-1).

Table B-1. *Numeric data types in Python*

Numeric Data Type	Representation
Integers	10, 254, 9876540
Floating-Point Numbers	5.7, 0.333, 0.5e6, 3.23e-5
Complex Numbers	4+8j

Arithmetic Operators

Of course, Python also provides arithmetic operators and some built-in functions for performing mathematical operations (Table B-2).

Table B-2. *Arithmetic operators and expressions*

Operator	Example
Addition/Subtraction	x + 5, 20 - j
Multiplication/Division	n ∗ 10, 5 / m (m ≠ 0)
Exponents	n ∗∗ 3
Modulo Operation	100 % d

413

Working with Numeric Data Types

Table B-3 lists a few of the built-in mathematical functions in Python.

Table B-3. *Python methods for working with numeric values*

Functions	Description
abs(value)	Returns the absolute value of a number.
max(iterable)	Returns the largest value in a list or other iterable object.
min(iterable)	Returns the smallest value in a list or other iterable object.
round(value)	Rounds a floating-point number and returns an int value.
sum(iterable)	Returns the sum of the items in a list or other iterable object.

String Data Type

Sequences of textual data can be represented using **strings**. Strings can be delimited with single or double quotes (Table B-4).

Table B-4. *String data type examples in Python*

Numeric Data Type	Representation
String Examples	'This is a string literal.' "String with numbers - 1234."
Empty String	'', ""
Escape Sequences in Strings	"Include quote \" character."
Raw Strings	r"Escape \nchars \tignored."

The escape character backslash, '\', can be used to include quotes or whitespace characters, such as '\n' for a newline or '\t' for a tab, in a string.

Including 'r' before the beginning of a string completely ignores all escape characters. Raw strings are very useful for regular expressions. Triple quotations can also be used to create multiline strings.

Workings with Strings

Strings in Python are **immutable** – these are objects that cannot be modified after they are created. Whenever you apply an operation to a string, a copy is created. Parts of a string, or substring, can also be accessed using indices and slices. The following bit of code looks at creating and accessing parts of strings:

```
a_string = "I like dogs." # Variable assignment
print(a_string[4]) # Select and print item at index 4
print(a_string[7:10]) # Print items from index 7 to index 9
```

Individual characters in a list can be accessed by selecting the index value associated with that character. The first item in a string has an index value of 0. Using a_string[4], the letter 'k' is selected. You can also start from the end of the string and work backward by using negative index values, starting from -1.

Use slicing to select a longer substring. In the preceding example, a_string[7:10] will include everything from a_string[7] to a_string[9], leaving out the final character in the string at index 10.

To test if a substring is or is not contained within a larger string, use the special membership operators, in and not in.

```
a_string = "Connor, Sarah"
print('Sarah' in a_string) # Returns and prints True
print('John' not in a_string) # Returns and prints True
```

It is also possible to manipulate the content of strings. Concatenation is the process of combining strings and can be achieved using the addition symbol, +.

```
first_name = "Peter"
last_name = "Parker"
full_name = first_name + " " + last_name # 'Peter Parker'
```

Finally, Table B-5 lists some of the other built-in methods for working with strings. Methods are similar to functions, except they are "called on" an object using specific keywords and a period, '.'. Methods such as isupper() and islower() are useful for testing and comparing strings and validating user input.

Table B-5. *More methods of the String class*

String Methods	Description
upper()	Converts all letters in the string to uppercase.
lower()	Converts all letters in the string to lowercase.
isupper()	Returns True if the string has at least one letter and all letters are uppercase.
islower()	Returns True if the string has at least one letter and all letters are lowercase.
isalpha()	Returns True if the string contains only letters and is not blank.
isalnum()	Returns True if the string contains only letters and numbers and is not blank.
isdecimal()	Returns True if the string contains only numeric characters and is not blank.
startswith(str, begin, end)	Returns True if the string or substring value begins with specified prefix.
endswith(str, begin, end)	Returns True if the string or substring value ends with specified prefix.
count(str, begin, end)	Returns the number of times a specified string occurs in a string or substring.
delimiter. join(iterable)	Concatenates a list of strings together into a single string value. Use the delimiter to specify how the individual strings are separated.
split(delimiter, maxsplit)	Takes a single string object and returns a list of strings. Use the delimiter to specify where in the string a split should occur.
strip(chars)	Removes specified chars and whitespaces from the beginning and the ending in a string.

Boolean Data Type

Objects in Python 3 can be assigned one of two Boolean values, True or False. Non-Boolean objects, ones that are assigned integer or string values, can also be evaluated using Boolean operators and expressions. This comes in handy in situations where you need to check if two strings match or to compare two integer values.

```
num_1, num_2 = 35, 10 # Ex of multiple assignment statement
print(num_1 == num_2) # Prints False
print(num_1 > num_2) # Prints True
```

Data Structures in Python

As you begin working with more data, you will need a way to organize, store, and access that information. Python includes a few data structures with their own special sets of rules – lists, tuples, sets, and dictionaries.

Lists

A **list** is a mutable data structure that is created using brackets:

```
# List with mixed data types
a_list = ["John Doe", 23, 1976, "blue"]
# Example of nested list
nested_list = [["zebra", "tiger", "turtle"], [3, 2, 7]]
```

Mutable objects are ones whose content can be modified without creating a new object. You can append, remove, or even rearrange items within the list. Similar to strings, items in lists are in an ordered sequence and can also be accessed using indices and slicing.

```
print(a_list[2]) # Access and print item at index 2
print(a_list[0:3]) # Only print items from indices 0 to 2
print(a_list[:]) # Print the entire list
```

To access the values inside nested lists

```
print(nested_list[0]) # Prints ['zebra', 'tiger', 'turtle']
print(nested_list[1][1]) # Prints 2
```

To get the length of a list or other iterative objects such as strings or tuples, use the built-in function len():

```
print(len(a_list)) # Prints 4
```

Similar to strings, lists and other iterables can also use the `in` and `not in` operators to check if values are present in the list.

```
print("John Doe" in a_list) # Prints True
```

Table B-6 contains some useful methods for working with lists.

Table B-6. *Some List class methods*

List Methods	Description
append(value)	Appends a single item to the end of a list.
count(value)	Returns the number of times a value occurs in a list.
index(value)	Returns the index of the first occurrence of a value in a list.
insert(index, value)	Inserts one value at a specific index location.
extend(iterable)	Adds more than one item to the end of a list.
pop(index)	Removes the item at the specified index.
remove(value)	Removes the first occurrence of a value in a list.
clear()	Clears the items in a list.
sort()	Sorts the items in a list.

It is also possible to remove items or an entire list using the `del` statement.

```
del a_list[3] # Delete item at index 3
# Delete multiple items using slicing
del a_list[0:2]
del a_list # Delete the entire list
```

Tuples

A **tuple** is an immutable data structure that is created using parentheses:

```
# Tuple with mixed data types
a_tuple = ("Jane Doe", 25, 1982, "brown")
```

It is possible to find values in a tuple using indices and slicing just like lists. However, since tuples are immutable, actions such as appending, sorting, or replacing items cannot be performed on them. This can be very useful when you have data that you may want others to view, but not have the ability to alter.

Many of Python's built-in functions such as `max()` and `len()` can also be used with tuple objects (Table B-7).

Table B-7. *Tuple class methods*

Tuple Methods	Description
index(value)	Returns the index of the first occurrence of a `value` in a tuple.
count(value)	Returns the number of times a `value` occurs in a tuple.

Sets

A **set** is a mutable data structure that is created using curly brackets:

```
# Set with mixed data types
a_set = {"John Smith", 45, 2001, "brown"}
```

Generally, sets are thought of as a collection of data of the same type, but they can also contain different data types. Sets in Python are unordered and unindexed and do not allow for duplicate values. Therefore, methods such as `index()` cannot be used with them. Sets can be very useful for removing repeated values and for performing mathematical operations from set theory on data. A few of those methods can be found in Table B-8.

Table B-8. *A few important Set class methods*

Set Methods	Description
add(value)	Adds a single item to a set.
update(iterable)	Adds multiple items to a set.
remove(value)	Removes the specified value from a set.
clear()	Empties all items in the set.
difference(set)	Returns the difference of two or more sets as a new set.
intersection(set)	Returns the intersection of two or more sets as a new set.
union(set)	Returns a set that contains the union of two or more sets.

Dictionaries

A **dictionary** is a mutable data structure composed of key/value pairs that is created using curly brackets:

```
# Example of dictionary
a_dict = {"name": "Jane Smith", "age": 29, "year": 1970, "eye color":
"green"}
```

In the preceding example, "name" is a **key** and "Jane Smith" is **value** associated with it. The key is similar to the index in lists. You can use the key to access and organize specific items in the dictionary.

Even though dictionaries are unordered, it is very easy to access their contents using the keys. To find out a key's value, look at the following bit of code:

```
print(a_dict["age"]) # Prints value associated with "age"
```

There are also a few methods that can help you to work with the items in dictionaries:

- keys() – Returns a list of all of the dictionary's keys

- values() – Returns a list of all of the dictionary's values

- items() – Returns a list containing a tuple for each key/pair value

For example, the following bit of code will return a list of all of the values in a_dict:

```
dict_values = list(a_dict.values()) # Create a list using typecasting
print(dict_values) # ['Jane Smith', 29, 1970, 'green']
```

It is also possible to check if keys or values exist in a dictionary using the membership operators in and not in. By default, if you do not specify keys or values, Python will search through the keys. The following example is equivalent to searching for "year" in a_dict.keys():

```
print("year" in a_dict) # Prints True
```

To add a new key/value pair to the dictionary, you could use an if statement to first check if the key already exists or not. If it does not, then add the new item to the dictionary.

```
if "height" not in a_dict:
    a_dict["height"] = 1.82
```

Another way to check if an item can be found in a dictionary is to use the setdefault() method. If the key does not already exist or does not have a value, then the key/value pair is added. However, if the key already exists and has a value, then the setdefault() method does not make any changes to the dictionary. The following code will add the "hair" key and its value to a_dict:

```
a_dict.setdefault("hair", "brown")
```

Table B-9 lists additional Dictionary class methods.

Table B-9. *Some other Dictionary class methods*

Dictionary Methods	Description
get(key, default value)	Returns the value of the specified key. If the key does not exist, then get() uses the default value. Useful for avoiding errors.
copy()	Creates a copy of a dictionary.
update({key: value})	Adds a new key/value pair to a dictionary.
pop(key)	Removes an item from the dictionary by specifying the key.
clear()	Clears all of the items from a dictionary.

Data Type Conversion

Typecasting is the action of directly converting one data type to another. In Python, it is possible to convert integers to floating-point numbers, integers to strings, lists to tuples, as well as other types of conversions (Table B-10). The following example shows how to convert a list into a dictionary:

```
info = [["name", "Sam"], ["age", 12]]
print(dict(info)) # {'name': 'Sam', 'age': 12}
```

Table B-10. *Typecasting functions*

Types	Typecasting Functions	Description
Arithmetic	int(value, base)	Converts different data types to int values. Specify the base if converting a string.
	float(value)	Converts integer to floating-point value.
	ord(character)	Converts a single character to an integer.
Strings	str(value)	Converts other data types into a string.
Data Structures	list(iterable)	Converts an iterable object into a list.
	tuple(iterable)	Converts an iterable object into a tuple.
	set(iterable)	Converts an iterable object into a set. May result in some data loss if there are duplicate values.
	dict(iterable)	Converts an iterable with structure (key, value) into a dictionary.

Conditionals and Loops in Python

The following section takes a look at programming tools that are used for controlling the flow and execution of commands and repetitive tasks while a program is running. Any of the following statements can be placed inside of another to create **nested loops**.

"if-elif-else" Conditional Statements

Conditional statements are used in programming to decide whether or not to perform certain actions based on whether the specified Boolean constraints evaluate to True or False. These types of statements are handled in Python using if-elif-else statements.

The following code shows a simple example of how to use if statements to check if items exist in a list:

```python
car_types = ["economy", "sedan", "convertible", "SUV", "economy"]
if "SUV" not in car_types:
    car_types.append("SUV")
elif car_types.count("economy") > 1:
    car_types.remove("economy")
else:
    car_types.append("luxury")
print(car_types) # ['sedan', 'convertible', 'SUV', 'economy']
```

The first if statement evaluates to False since "SUV" already exists in the car_types list. The elif statement provides other conditions to check if previous clauses were not true. Finally, else statements are executed if all previous conditions were false.

There are other operators besides in and not in that can be used for evaluating Boolean expressions. Table B-11 lists some of them.

Table B-11. *Python operators*

Types	Operators	Description
Comparison	`x == y`	Equal to
	`x != y`	Not equal to
	`x > y`	Greater than
	`x < y`	Less than
	`x >= y`	Greater than or equal to
	`x <= y`	Less than or equal to
Logical	`x > 5 and y < -5`	Returns `True` if both statements are true.
	`x > 5 or y < -5`	Returns `True` if both statements or only one of them is true.
	`not(x > 5)`	Returns the opposite. Changes `True` to `False` and vice versa.

"for" Loops

Python is very useful for automating repetitive tasks using loops. The `for` loop is very useful when you have a task that needs to be executed a certain number of times, such as traversing through a sequence until you reach the end.

```
colors = ["blue", "red", "purple", "green", "white"]
for color in colors:
    print("Current color: {}".format(color))
```

The preceding code will cycle through the list five times, once for each item in the list. There are also ways to repeat a block of code or iterate over a sequence of numbers using the `range()` function. The following code will print out the index values and their corresponding colors from the `colors` list:

```
for i in range(len(colors)):
    print("{}. {}".format(i + 1, colors[i]))
```

The enumerate() function is also a way to include a counter while iterating through a sequence.

```python
for count, color in enumerate(colors, start=1):
    print("{}. {}".format(count, color))
```

List Comprehensions

Now that we have gone over for loops and if statements, let's take a look at a very important concept in Python, **list comprehensions**. List comprehensions are a way to create new lists from more compact code.

Given a list of strings and integers, the following example uses list comprehension to create a new list of only the string values:

```python
a_list = ["Sam", 1978, "Elsa", 1984, "Marcus", 1980, "Trevor", 1983]
new_list = [word for word in a_list if type(word) == str]
print(new_list) # ['Sam', 'Elsa', 'Marcus', 'Trevor']
```

Using nested loops, the following code is equivalent to the preceding example:

```python
new_list = [] # Creates an empty list
for word in a_list:
    if type(word) == str:
        new_list.append(word)
print(new_list) # ['Sam', 'Elsa', 'Marcus', 'Trevor']
```

"while" Loops

Unlike the for loop, the while loop can be used to execute a block of code an unknown number of times, as long as the Boolean condition being tested continues to evaluate to True. When the computer reaches the end of a while clause, it returns back to the beginning of the loop and checks again if the condition is still true. If so, the clause is executed once more, and the condition will be checked again at the end.

```python
while True:
    print("Please enter your age: ")
    age = int(input())
    if age < 21:
```

```
        print("You are underage.")
        continue
    else:
        break
print("Access granted.")
```

For this example, the `while` condition will always evaluate to True. However, you could also use other logical or comparison operators to test different conditions. Once in the loop, the user is asked to input their age. This information is then handled using an `if-else` conditional statement.

In addition, there are types of statements that can be used to control the flow of the program and handle exceptions in a while loop – `break` and `continue`. These statements can also be used in `for` loops.

- `break` – Used to immediately exit a `while` loop. After exiting, the program will continue with the statement following the loop.

- `continue` – Used to immediately skip the rest of the loop and return back to start of the `while` loop. The condition statement is then reevaluated with the next iterative value.

Functions

A **function** is a collection of statements that perform some particular task and can be reused multiple times throughout a program. Functions also make code easier to read by avoiding duplicate code.

You can also define your own functions in Python using the `def()` statement. When you call a function by invoking its name, you can also pass values, known as arguments, between the parentheses. Since functions in Python are treated as objects, you can also pass them as arguments to other functions. They also always return values whether it is None or some calculated value. Functions can even return other functions.

```
def check_for_nums(items_list):
    """
    Checks to see if string (item) contains only
    letters. If the string contains a numeric value,
    it is removed from the list.
    """
```

```
    for item in items_list[:]:
        if item.isalpha() != True:
            items_list.remove(item)
    return items_list # Return statement and value

test_list = ["horse", "1234", "d0g", "mouse", "m3"]
new_list = check_for_nums(test_list) # Function call
print(new_list) # Prints ['horse', 'mouse']
```

The function check_for_nums() is used to iterate through a list and remove any strings that contain numeric values. The function takes as an argument a list of string values. When items_list is called in the for loop, we need to iterate over a copy of the list (created using [:]) rather than the actual list itself in order to modify it using remove(). The for loop could also be written more concisely using list comprehension.

Be sure to consider local and global scope when creating functions and passing arguments between them. Variables and values created in a called function only exist within that function's **local scope**. Other variables and parameters created outside of all functions exist in the **global scope** and can be accessed by all functions.

Lambda Functions

Lambda functions are often referred to as anonymous functions, or functions that have not been assigned a name. They are single expressions that can take multiple arguments. The general form for a lambda function is

```
lambda arguments : expression
```

The following example demonstrates how to create and call a multiargument lambda function in Python:

```
full_name = lambda f, l: "Full name: {} {}".format(f, l)
print(full_name("Ben", "Franklin"))
```

This is equivalent to creating a function using def().

```
def full_name(first, last):
    print("Full name: {} {}".format(first, last))
```

Lambda functions are very useful when creating GUIs for mapping actions in response to events. When a button is clicked and a signal is triggered, a method, also known as a slot, handles that event. A method could be the call to a function, but in some instances, a lambda function can be used for the same reason.

Object-Oriented Programming (OOP)

In this section we will take a brief look at what object-oriented programming is and how to create objects using classes.

There are a number of different **programming paradigms**, or methods and styles of programming. One common method is **procedural programming** where a computer follows a sequential set of commands to perform some task. This kind of programming can make it difficult for adding new functions and working with more dynamic situations.

Another approach is **object-oriented programming (OOP)** which focuses on creating objects with their own properties and behaviors, and modeling their relationships between other objects. Objects are created using classes – which act as templates for the data, attributes, and methods that can be applied to an instance of a class. An **instance**, or an object created from a class, has access to all the data and methods inside the class.

OOP also introduces the idea of **inheritance**, which is the concept of creating new classes that derive properties and behaviors from existing classes. An object created using inheritance is known as a **child**. A child object has its own set of attributes and also acquires all of the properties and behaviors from its **parent** class.

The following example demonstrates how to create a new class, `Window`, for GUI development which inherits from the PyQt5 class, `QMainWindow`:

```python
# Import necessary modules
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow

class Window(QMainWindow):
    """
```

```
    Create a new class, Window, which inherits from other PyQt modules,
    in this case from the QMainWindow class in the QtWidgets module.
    Inheriting from QMainWindow means we have access to all the attributes
    to create GUIs using PyQt, but we also can add our own variables and
    methods in our new class.
    """

    def __init__(self):
        super().__init__() # super is used to access methods from parent
class
        self.initializeUI()

    def initializeUI(self):
        """
        Initialize the window and display its contents to the screen.
        """
        self.setGeometry(100, 100, 300, 200)
        self.setWindowTitle("Create the Window Class")
        self.show()

app = QApplication(sys.argv) # Create instance of QApplication class
window = Window() # Create instance of Window class
sys.exit(app.exec_()) # Start the event loop
```

The `window` object is an instance of the `Window` class. To make code easier to understand, classes start with an uppercase letter. The child object, `window`, inherits from both `Window` and from the `QMainWindow` class and is able to use the methods found in `QMainWindow`, such as `setGeometry()` and `show()`, to create our GUI window.

In order to reference the current class, the keyword `self` allows us to use any of the data or methods within `QMainWindow` as well as any new methods we create in the `Window` class.

Exception Handling in Python

It is quite common for programs to run into errors when they are running. **Syntax errors** are the ones you will most often run into when writing your code. They are caused by incorrect syntax such as forgetting to include a colon, ':', after a `for` clause.

Exceptions are the errors that occur when executing code. It is important to consider how your program will handle these exceptions rather than allowing your application to crash. Python includes a number of built-in exceptions, including how to handle not being able to locate or open a file, incorrect data types being passed into a function, dividing by zero, and various other situations.

The best way to handle an exception is to use the `try` and `except` statements. The following code shows how to check if a user correctly enters an integer value for their age:

```
while True:
    try:
        weight = int(input("Enter your weight (in lbs): "))
        break
    except ValueError:
        print("Invalid input.")
```

Code that could possibly cause an error is placed in the `try` clause. If an exception does occur while the program is running, then the `except` clause will catch it and handle the error accordingly. Depending upon the situation, you could display an error message, use the `break` or `continue` statements, or force a specific kind of error to occur.

Reading and Writing to Files in Python

Python also provides built-in modules and functions to open, append, and write to files. Working with and locating files in directories can be a very long topic that includes filename pattern matching, traversing directories, deleting files and directories, and more. This section will simply focus on the basics of opening and writing to files.

The simplest way to open a file is to use

```
with('path to file', mode) as f
```

There are other ways to open files, but using this pattern allows for cleaner code and ensures that files are attended to by closing the file once the function using the resource is finished executing.

```
with open ('quotes.txt', 'w') as f:
    text = "Life is what happens when you're busy making other plans. -John
    Lennon"
    f.write(text)
```

The open() method takes as arguments the path to the file and a mode to tell the computer how to handle the file. The second argument, 'w', means we want to write to a file and ensures that a new file is created if it does not exist already. The text is then written to a new file using f.write(). Other options to interact with files include reading the contents of an entire file with f.read() or only reading one line at a time using f.readline().

Table B-12 lists the common modes for reading and writing to files.

Table B-12. *Some modes for working with files*

Character	Definition
'r'	Opens a file for reading. This is the default mode.
'w'	Opens a file for writing. Overwrites a file if one already exists. Otherwise, creates a new file.
'a'	Opens a new file for appending. File pointer starts at the end of the file if it exists. Otherwise, creates a new file.
'r+'	Opens a file for reading and writing.
'w+'	Opens a file for writing and reading. Overwrites a file if one already exists. Otherwise, creates a new file.
'a+'	Opens a new file for appending and reading. File pointer starts at the end of the file if it exists. Otherwise, creates a new file.

Summary

Hopefully this appendix helps you to recall some method you may have forgotten about, or sparks an idea in your mind about how to solve a problem when creating your own GUIs, or maybe even helps you to learn something new about programming in Python.

This appendix covers information from data types and data structures to conditional statements and iteration using loops, to creating your own functions and classes, and more. But it only skims the surface of the possibilities of what you could apply in your own applications.

There are so many possibilities for working with PyQt and Python, and covering all of that information could definitely be written in more than one book. If you ever get stuck, the Internet is definitely an amazing resource full of information and guidance to working through your problems.

Best of luck in all of your projects and in your endeavors!

Index