Listing 3-6. Code for clearText() and acceptText() slots

```
# line_edits.py
 def clearText(self):
 """Clear the QLineEdit input field."""
 self.name_edit.clear()

 def acceptText(self):
 """Accept the user's input in the QLineEdit
 widget and close the program."""
 print(self.name_edit.text())
 self.close()
```

When clear_button is clicked, it emits a signal that is connected to the clearText() slot, and the name_edit widget will react to the signal and clear its current text. If the user clicks accept_button, the text in name_edit is read using the getter text() and printed in your computer's shell. The application then closes.

Let's take a look at another commonly found widget in desktop applications.

The QCheckBox Widget

The QCheckBox widget is a selectable button that generally has two states: on and off. This makes them 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 application in Figure 3-3 shows a basic questionnaire GUI. The user is allowed to select all checkboxes that apply to them, and each time the user clicks a checkbox, we call a method to show how to determine the widget's current state.

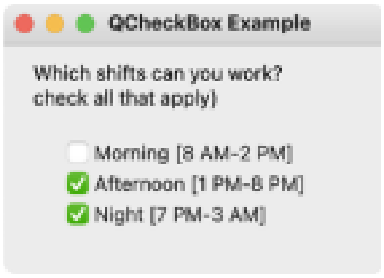


Figure 3-3. *Example that uses QCheckBox widgets*

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 or consider using QRadioButton.

The QCheckBox class can also be used in dynamic applications, where a series of checkbox widgets could be used to select or change a GUI's text, appearance, or even state (by enabling or disabling interactivity).

Explanation for Using QCheckBox

Begin creating the MainWindow class like before by using the empty window script from Chapter 1 as a template. For this application, import QCheckBox and other classes shown in Listing 3-7.

Listing 3-7. Setting up the main window for using QCheckBox widgets

```
# checkboxes.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QWidget, QCheckBox,
QLabel)
from PyQt6.QtCore import Qt

class MainWindow(QWidget):

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

 def initializeUI(self):
 """Set up the application's GUI."""
 self.setGeometry(100, 100, 250, 150)
 self.setWindowTitle("QCheckBox Example")
```

```
      self.setUpMainWindow()
      self.show()
```

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

In Listing 3-8, we set up the method for arranging widgets in the

window. ***Listing 3-8.*** The setUpMainWindow() method for using

checkboxes

```
# checkboxes.py
 def setUpMainWindow(self):
 """Create and arrange widgets in the main window."""  header_label
= QLabel("Which shifts can you work? \  (Please check all that
apply)", self)  header_label.setWordWrap(True)
 header_label.move(20, 10)

 # Set up the checkboxes
 morning_cb = QCheckBox("Morning [8 AM-2 PM]", self)
morning_cb.move(40, 60)
 #morning_cb.toggle() # Uncomment to start checked
morning_cb.toggled.connect(self.printSelected)

 after_cb = QCheckBox("Afternoon [1 PM-8 PM]", self)
after_cb.move(40, 80)
 after_cb.toggled.connect(self.printSelected)

 night_cb = QCheckBox("Night [7 PM-3 AM]", self)
 night_cb.move(40, 100)
 night_cb.toggled.connect(self.printSelected)
```

A QLabel widget is used to display a question to the user, helping the user to understand the purpose of the GUI. For labels with longer text that won't fit on one line, use the setWordWrap() method.

Three QCheckBox objects are also created, each with a variable name that is representative of the widget's purpose. The text displayed next to each checkbox is passed as the first argument. The QCheckBox method toggle() can be used to toggle

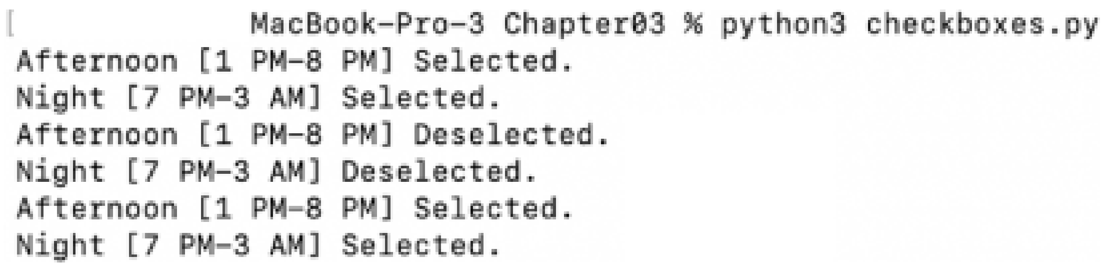
the checkbox on or off. When a checkbox is selected, rather than using the clicked signal like with QPushButton, use toggled to emit a signal that is connected to the slot printSelected(), shown in Listing 3-9.

Tip It is possible to connect multiple signals to the same slot.

Listing 3-9. Code for the printSelected() slot

```
# checkboxes.py
 def printSelected(self, checked):
  """Print the text of a QCheckBox object when selected or
deselected. Use sender() to determine which widget is sending the
signal."""
 sender = self.sender()
 if checked:
 print(f"{sender.text()} Selected.")
 else:
 print(f"{sender.text()} Deselected.")
```

The toggled() signal also passes along additional information, checked, which returns True if the checkbox is selected. Otherwise, it returns False. With so many widgets connected to the same slot, it can be hard to determine which widget is being interacted with and emitting the signal. Thankfully, the QObject method sender() returns which object (the widget) is sending the signal. (All widgets inherit the QObject class.) For this example, use the getter text() to get the checkbox object's text and print its value in the shell. An example of the output to the terminal can be seen in Figure 3-4.



```
[              MacBook-Pro-3 Chapter03 % python3 checkboxes.py
 Afternoon [1 PM-8 PM] Selected.
 Night [7 PM-3 AM] Selected.
 Afternoon [1 PM-8 PM] Deselected.
 Night [7 PM-3 AM] Deselected.
 Afternoon [1 PM-8 PM] Selected.
 Night [7 PM-3 AM] Selected.
```

Figure 3-4. *Output to the shell when the different checkboxes are selected or deselected*

Let's take a look at one more very important class for creating interactive and user friendly GUIs.

The QMessageBox Dialog

When a user closes an application or saves their work, or an error occurs, they will typically see a dialog box 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. Dialog boxes are a very important form of **feedback**, or methods of monitoring and communicating changes back to the user.

The QMessageBox class can be used to not only alert the user to a situation but also to decide how to handle the matter. For example, when closing a document you just modified, you might get a dialog box with buttons asking you to Save, Don't Save, or Cancel. Four common types of predefined QMessageBox widgets in PyQt are shown in Table 3-1.

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

QMessageBox Icons	Types	Details
❓	Question	Ask the user a question
ℹ	Information	Display information during general operations

⚠ Warning Report noncritical errors ✖ Critical
Report critical errors

Windows vs. Dialogs

Applications will typically consist of one main window. A **window** is used to visually separate applications from each other and generally consists of menus, a toolbar, and other kinds of widgets that can often act as the main interface for a GUI application. Windows in Qt are typically considered widgets that appear on the screen and don't have a parent widget.

A **dialog box**, or simply **dialog**, pops up and displays 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 dialogs can be used to update the main window.

There are two kinds of dialog boxes. **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.

Explanation for Using QMessageBox

The QMessageBox class produces a modal dialog box, and in the following example, we will take a look at how to use three of the predefined QMessageBox message types: question, information, and warning.

Note For this example, you will also need the authors.txt file found in the files folder of this chapter's repository on GitHub.

This application's main window can be seen in Figure 3-5, and a couple of QMessageBox dialogs are shown in Figure 3-6.

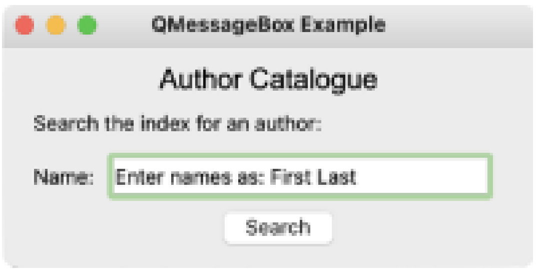


Figure 3-5. *Main window for the QMessageBox example where the user can search for an author's name in a text file*

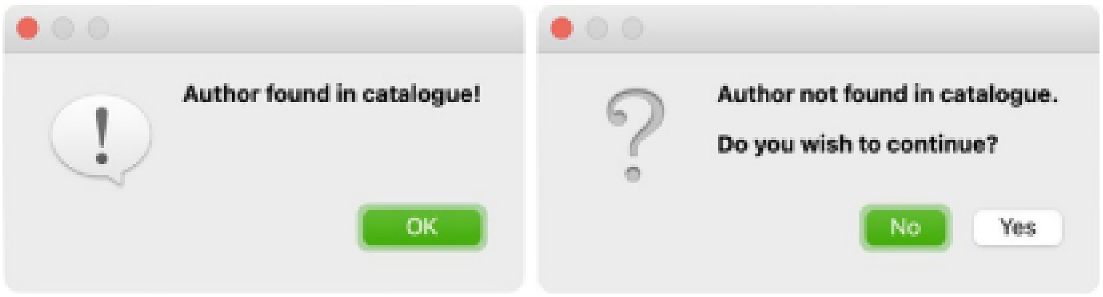


Figure 3-6. *Information dialog (left) that lets the user know that their search was successful. Question dialog (right) that asks if the user wants to continue searching if the author wasn't found*

For Listing 3-10, you will need to import a few additional classes, including the QMessageBox class from QtWidgets, into the empty window script from Chapter 1.

Listing 3-10. Setting up the main window for using QMessageBox dialogs

```
# message_boxes.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QWidget, QLabel,
QMessageBox, QLineEdit, QPushButton)
from PyQt6.QtGui import QFont

class MainWindow(QWidget):

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

```
    def initializeUI(self):


    """Set up the application's GUI."""
    self.setGeometry(100, 100, 340, 140)
    self.setWindowTitle("QMessageBox Example")

    self.setUpMainWindow()
    self.show()

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

To build the window seen in Figure 3-5, you'll need to create a few QLabel objects, a QLineEdit widget for the user to enter an author's name, and a QPushButton object that emits a signal when pressed and searches for the text in a text file. This is all handled in Listing 3-11.

Listing 3-11. The setUpMainWindow() method for using message boxes

```
# message_boxes.py
 def setUpMainWindow(self):
 """Create and arrange widgets in the main window."""
```

```
catalogue_label = QLabel("Author Catalogue", self)
catalogue_label.move(100, 10)
 catalogue_label.setFont(QFont("Arial", 18))

 search_label = QLabel(
 "Search the index for an author:", self)
 search_label.move(20, 40)

 # Create author QLabel and QLineEdit widgets
 author_label = QLabel("Name:", self)
 author_label.move(20, 74)

 self.author_edit = QLineEdit(self)
 self.author_edit.move(70, 70)
 self.author_edit.resize(240, 24)
 self.author_edit.setPlaceholderText(
 "Enter names as: First Last")
```

```
# Create the search QPushButton
search_button = QPushButton("Search", self)
search_button.move(140, 100)
search_button.clicked.connect(self.searchAuthors)
```

The widgets catalogue_label and search_label are used to convey information to the user. In PyQt, QLabel widgets are often placed next to QLineEdit and other input widgets as tags. The labels can then be linked to the input widgets as **buddies**. Here, the author_label and author_edit are simply placed next to each other.

Placeholder text can be used to give the user extra information about a QLineEdit widget's purpose or for how to format input text. This is done with setPlaceholderText(). An example of this is seen in Figure 3-5.

Lastly, search_button emits a signal that calls the slot searchAuthors() in Listings 3-12 and 3-13.

Listing 3-12. First part of the code for the searchAuthors() slot

```
# message_boxes.py
 def searchAuthors(self):
 """Search through a catalogue of names.
 If a name is found, display the Author Found dialog.  Otherwise,
display Author Not Found dialog."""
 file = "files/authors.txt"

 try:
 with open(file, "r") as f:
 authors = [line.rstrip("\n") for line in f]

 # Check for name in authors list
 if self.author_edit.text() in authors:
 QMessageBox.information(self, "Author Found",  "Author found in
catalogue!",
 QMessageBox.StandardButton.Ok)
```

When the user clicks on search_button, the program will try to open the authors. txt file and store its contents in the authors list. If the user enters a name in author_ edit that is contained in the authors.txt file, an information dialog appears like the first image in Figure 3-6.

To create a QMessageBox dialog from one of the predefined types, first, create a QMessageBox object and call one of the static functions, in this case, information. Next, pass the parent widget. Then set the dialog's title, "Author Found", and the text that will appear inside the dialog that provides feedback, possibly with information about actions a user could take. This is followed by the types of standard buttons that will appear in the dialog. Multiple buttons can be used and separated with a pipe key, |. Standard buttons include Open, Save, Cancel, Reset, Yes, and No. The Appendix lists other QMessageBox. StandardButton types.

Note   On macOS, when a message box appears, the title is generally ignored due to macOS Guidelines. If you are using an Apple computer and don’t see a title in the dialog boxes, don’t fear! You haven’t done anything wrong.

Listing 3-13. Second part of the code for the searchAuthors() slot

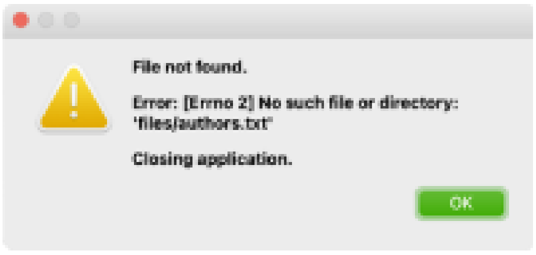
```
# message_boxes.py
 else:
 answer = QMessageBox.question(self,
 "Author Not Found",
 """<p>Author not found in catalogue.</p>  <p>Do you wish to
continue?</p>""",
 QMessageBox.StandardButton.Yes | \
 QMessageBox.StandardButton.No,
 QMessageBox.StandardButton.No)

 if answer == QMessageBox.StandardButton.No:  print("Closing
application.")
 self.close()
 except FileNotFoundError as error:
 QMessageBox.warning(self, "Error",
 f"""<p>File not found.</p>
```

```
 <p>Error: {error}</p>
 Closing application.""",
 QMessageBox.StandardButton.Ok)
 self.close()
```

If the author is not found, a question dialog (second image in Figure 3-6) appears asking the user if they want to search again or quit the program. The standard buttons Yes and No appear in the window. The final argument is used to specify which button you want to highlight in the dialog and set as the default button.

Note PyQt text widgets are able to display rich text using a subset of the HyperText Markup Language (HTML) and Cascading Style Sheets (CSS). This topic is explored more in Chapter 6, but for now, we'll use HTML to arrange the text that is placed between the HTML tags <p> and </p> into paragraphs.

If an error occurs and the file is not found, the warning dialog in Figure 3-7 appears.


File not found.

Error: [Errno 2] No such file or directory: 'files/authors.txt'

Closing application.

Figure 3-7. *Warning dialog that informs the user that the authors.txt file was not found*

With everything you have learned up to this point, it is a good time to practice creating a larger project.

Project 3.1 – Login GUI and Registration Dialog

A login user interface is probably one of the most common interfaces you interact with on a regular basis – signing into your computer, your online bank account, or your email or social media accounts; logging into your phone; or even signing up for some new app. The login interface is everywhere.

For this example, you will create three different windows. The first window that will appear is the login GUI in Figure 3-8.

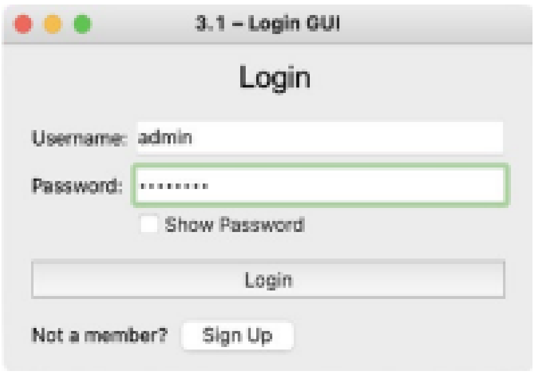
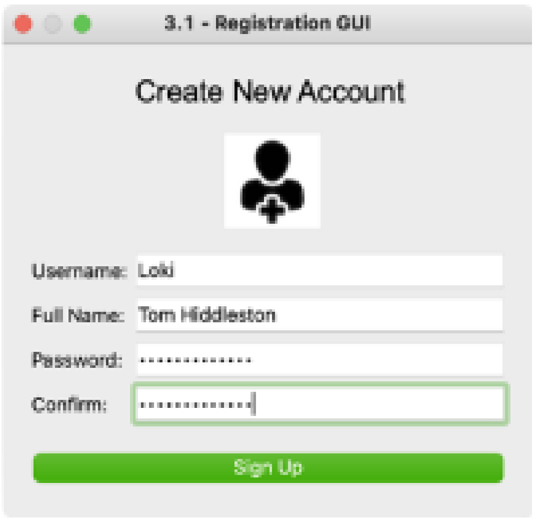


Figure 3-8. *The Login window*

This project also demonstrates how to open and close other windows and dialogs and begins looking at how to use event handlers. The first time someone uses your applications, you may want them to sign up and create their own username and password. The Registration dialog in Figure 3-9 appears when a user clicks the Sign Up button in Figure 3-8.



Username: Loki
Full Name: Tom Hiddleston
Password: ••••••••••••••
Confirm: •••••••••••••

Figure 3-9. *The Registration dialog for creating a new user*

If a user successfully logs in, they will be greeted to the window in Figure 3-10 that simply displays a QLabel widget with the image of a kingfisher.

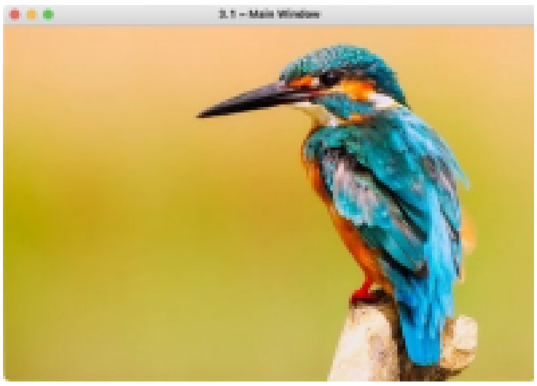


Figure 3-10. *The application's main window that appears if a user successfully logs in. The image of the kingfisher is from*

https://pixabay.com

The next section discusses the different windows, widgets, and their functionalities in this application.

Designing the Login GUI and Registration Dialog

When designing a login interface, you want to create a GUI that clearly labels its widgets, differentiates between the login and signup fields, and helps users to better navigate through potential errors, such as if caps lock is on or if the username is incorrect. 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 when checked
- Buttons that users can click to log in or even register new accounts

For the Login window in Figure 3-8, two separate QLineEdit widgets are used for users to enter their username and password. Under the password QLineEdit widget, there is a checkbox to toggle if the password is hidden or not. There are also two QPushButton widgets: one that the user can click to log in and the other to register a new account.

When the user clicks the Login button, a signal is emitted. The connected slot is used to check if input is correct. QMessageBox dialogs are used to provide feedback if the user exists or does not exist, or as an error message if the users.txt file does not exist. If a successful login does occur, then the main window in Figure 3-10 will appear. You can find users.txt in the files folder of this chapter's GitHub repository.

If a user wants to register a new account, they can click the Sign Up button in the Login window and the *modal* dialog in Figure 3-9 appears. The user cannot interact with the Login window unless the Registration dialog is closed.

Finally, this example also demonstrates how to handle the event when the user closes the window. Rather than just closing the application with close(), you will see how to use the event handler closeEvent() to customize how your programs can close. The Login GUI is what the user first sees, so let's begin there.

Explanation for Creating the Login GUI

The widgets and concepts learned throughout Chapters 1, 2, and 3 will be applied in this project. Listing 3-14 starts the project by using the empty window script from Chapter 1 to begin building the LoginWindow class. Go ahead and import many of the widget and other classes you have seen before as well as the NewUserDialog class you'll create later in the "Explanation for Creating the Registration Dialog" section.

Listing 3-14. Setting up the window for the Login GUI

```
# login_gui.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QWidget, QLabel,
QLineEdit, QPushButton, QCheckBox, QMessageBox)
from PyQt6.QtGui import QFont, QPixmap
from PyQt6.QtCore import Qt
from registration import NewUserDialog

class LoginWindow(QWidget):

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

```
def initializeUI(self):
"""Set up the application's GUI."""
self.setFixedSize(360, 220)
self.setWindowTitle("3.1 – Login GUI")

self.setUpWindow()
```

```
 self.show()

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

The LoginWindow class is not this application's main window. The class could inherit the QDialog class, the base class for creating dialogs, but we don't need to worry about window modality here, so simply inheriting QWidget will work. Also, setFixedSize() is used to stop the window from growing or shrinking.

Listing 3-15 begins setting up the setUpWindow() method for the LoginWindow

class. ***Listing 3-15.*** First part of the setUpWindow() method for the Login

GUI

```
# login_gui.py
 def setUpWindow(self):
 """Create and arrange widgets in the main window."""
self.login_is_successful = False

 login_label = QLabel("Login", self)
 login_label.setFont(QFont("Arial", 20))
 login_label.move(160, 10)

 # Create widgets for username and password
 username_label = QLabel("Username:", self)
 username_label.move(20, 54)

 self.username_edit = QLineEdit(self)
 self.username_edit.resize(250, 24)
 self.username_edit.move(90, 50)
```

```
    password_label = QLabel("Password:", self)
    password_label.move(20, 86)
```

```
self.password_edit = QLineEdit(self)
self.password_edit.setEchoMode(
QLineEdit.EchoMode.Password)
self.password_edit.resize(250, 24)
self.password_edit.move(90, 82)
```

The login_is_successful variable keeps track of whether or not the user has logged in. A couple of QLabel and QLineEdit input widgets are created for entering a username and a password. The widgets are then arranged side by side. The setEchoMode() method provided by QLineEdit is very useful for hiding text as it is being input. The Password flag is used here to mask characters while entering the password. Listing 3-16 continues creating and arranging widgets in LoginWindow.

Listing 3-16. Second part of the setUpWindow() method for the Login GUI

```
# login_gui.py
# Create QCheckBox for displaying password
self.show_password_cb = QCheckBox(
"Show Password", self)
self.show_password_cb.move(90, 110)
self.show_password_cb.toggled.connect(
self.displayPasswordIfChecked)

# Create QPushButton for signing in
login_button = QPushButton("Login", self)
login_button.resize(320, 34)
login_button.move(20, 140)
login_button.clicked.connect(self.clickLoginButton)

# Create sign up QLabel and QPushButton
not_member_label = QLabel("Not a member?", self)
not_member_label.move(20, 186)

sign_up_button = QPushButton("Sign Up", self)
sign_up_button.move(120, 180)
sign_up_button.clicked.connect(self.createNewUser)
```

Create a QCheckBox called show_password_cb that, when toggled, will emit a signal that calls the displayPasswordIfChecked() slot. The login_button uses the clicked signal to call the clickLoginButton() slot. The sign_up_button opens up a dialog to register new users and calls the createNewUser() slot when clicked.

The next step is to create the various slots. We'll start in Listing 3-17 with clickLoginButton() used by login_button.

Listing 3-17. First part of the clickLoginButton() slot

```
# login_gui.py
 def clickLoginButton(self):
 """Check if username and password match any existing  entries in
users.txt.
 If found, show QMessageBox and close the program.  If they
don't, display a warning QMessageBox."""  users = {} #
Dictionary to store user information  file = "files/users.txt"

 try:
 with open(file, "r") as f:
 for line in f:
 user_info = line.split(" ")
 username_info = user_info[0]
 password_info = user_info[1].strip("\n")  users[username_info] =
password_info

 # Collect user and password information
 username = self.username_edit.text()
 password = self.password_edit.text()
```

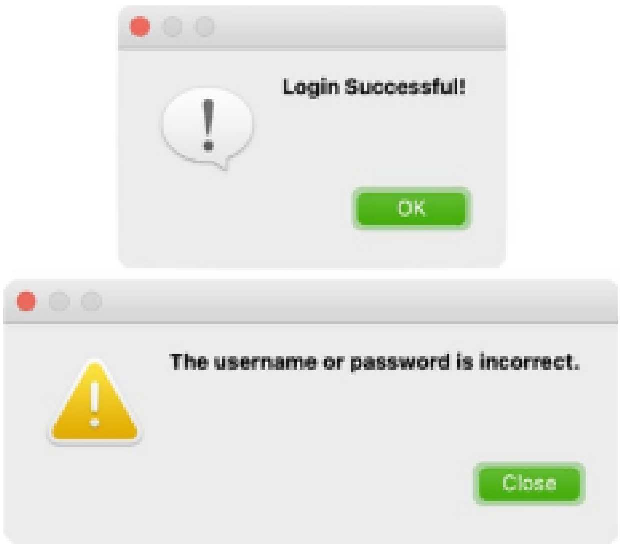
This method, which is continued in Listing 3-18, first checks to see if the users.txt file exists. If it does, the user's information is collected from the file, and username_info and password_info values are added to the users dictionary. Next, the text values for username_edit and password_edit are collected using text().

Listing 3-18. Second part of the clickLoginButton() slot

```
# login_gui.py
 if (username, password) in users.items():
 QMessageBox.information(self,
 "Login Successful!",
 "Login Successful!",
 QMessageBox.StandardButton.Ok,
 QMessageBox.StandardButton.Ok)
 self.login_is_successful = True
 self.close() # Close the login window
 self.openApplicationWindow()
 else:
 QMessageBox.warning(self, "Error Message",  "The username or
password is incorrect.",  QMessageBox.StandardButton.Close,
 QMessageBox.StandardButton.Close)
 except FileNotFoundError as error:
 QMessageBox.warning(self, "Error",
 f"""<p>File not found.</p>
 <p>Error: {error}</p>""",
 QMessageBox.StandardButton.Ok)
 # Create file if it doesn't exist
 f = open(file, "w")
```

The Python dict method items() returns a list of key-value pairs as tuples that can be used to check for a matching username-password pair in users.

If a match is found, the top QMessageBox in Figure 3-11 pops up. Then login_is_ successful is set to True, and the current window closes. The example’s main window appears by calling openApplicationWindow(), which is created in the “Explanation for Creating the Main Window” section. It is worth noting that close() does not actually close the window like you may think. The window is merely hidden from view. This is explored further in the “Using Event Handlers to Close a Window” subsection.



Login Successful!

The username or password is incorrect.

Figure 3-11. *The information dialog (top) that informs the user that their input was correct. The warning dialog (bottom) that informs the user of an error*

Otherwise, a warning QMessageBox, the bottom image in Figure 3-11, is displayed if the username or password is incorrect.

The following subsections finish creating the LoginWindow class.

Hiding Input for QLineEdit

The toggled signal used by show_password_cb in Listing 3-16 is connected to the displayPasswordIfChecked() slot in Listing 3-19.

Listing 3-19. Code for the displayPasswordIfChecked() slot

```
# login_gui.py
 def displayPasswordIfChecked(self, checked):
 """If QCheckButton is enabled, view the password.  Else, mask the
password so others can not see it."""  if checked:
 self.password_edit.setEchoMode(
 QLineEdit.EchoMode.Normal)
 elif checked == False:
 self.password_edit.setEchoMode(
 QLineEdit.EchoMode.Password)
```

If show_password_cb is checked, then the password's characters can be seen using setEchoMode() and passing the enum QLineEdit.EchoMode with the Normal flag. Otherwise, if unchecked, the password's text is masked so others cannot see the characters. An example of this can be seen in Figure 3-8. Flags also exist for hiding the password completely, NoEcho, and for displaying only the character being entered and masking others, PasswordEchoOnEdit.

How to Open a New Window or Dialog

It is possible to have multiple windows and dialogs open at the same time in PyQt. Opening a QMessageBox is relatively easy – simply create a QMessageBox instance when needed. However, for custom dialogs and windows, you will also need to call a method to display them.

The sign_up_button, when clicked, emits a signal that is connected to the createNewUser() method seen in Listing 3-20.

Listing 3-20. Code for the createNewUser() slot and openApplicationWindow() method

```
# login_gui.py
 def createNewUser(self):
 """Open a dialog for creating a new account."""
 self.create_new_user_window = NewUserDialog()
 self.create_new_user_window.show()

 def openApplicationWindow(self):
 """Open a mock main window after the user logs in."""
self.main_window = MainWindow()
 self.main_window.show()
```

In createNewUser(), create an instance of NewUserDialog from the registration module. To display the modal dialog, call show(). Take a look at the "Explanation for Creating the Registration Dialog" section for creating the class. It is a similar pattern for opening the main window after the user logs in.

Using Event Handlers to Close a Window

A good practice when quitting a program is to present a dialog box, like the one in Figure 3-12, 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.

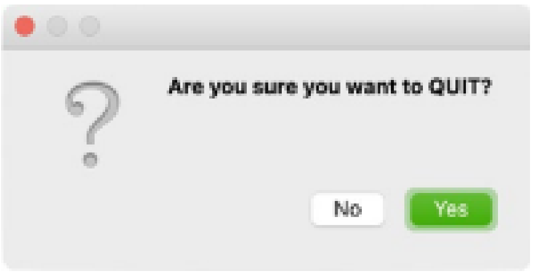


Figure 3-12. *QMessageBox that appears before quitting the application*

When an event occurs, an event object is created depending upon the type of event, and that object is passed to the appropriate object, such as a widget. Event handlers are then used to take care of the event if accepted. Otherwise, the event can be ignored.

When a QWidget is closed in PyQt, a QCloseEvent object is generated. However, widgets and windows are not actually closed. Rather, they are hidden from view if the event is accepted. The reason that the application actually quits when the LoginWindow instance is closed is due to a signal that is emitted when the last main window (one with no parent) is no longer visible. That signal is QApplication.lastWindowClosed(), which is already handled by PyQt.

In order to change how the closeEvent() method is handled, in this case, for the LoginWindow class, you will need to reimplement the closeEvent(). An example of this is shown in Listing 3-21.

Listing 3-21. Modifying the closeEvent() event handler

```
# login_gui.py
 def closeEvent(self, event):
```

```
"""Reimplement the closing event to display a
QMessageBox before closing."""
if self.login_is_successful == True:
event.accept()
else:
```

```
 answer = QMessageBox.question(
 self, "Quit Application?",
 "Are you sure you want to QUIT?",
 QMessageBox.StandardButton.No | \
 QMessageBox.StandardButton.Yes,
 QMessageBox.StandardButton.Yes)
 if answer == QMessageBox.StandardButton.Yes:
 event.accept()
 if answer == QMessageBox.StandardButton.No:
 event.ignore()
```

If the window was closed after login was successful, the event is accepted using the accept() method. If the window is closed for some other reason, a QMessageBox asks the user if they are sure about quitting. If the response from the question QMessageBox is Yes, the close event is accepted, and the program is closed. Otherwise, the event is ignored using ignore().

The next section creates the MainWindow class.

Explanation for Creating the Main Window

The main window used in this project is a very basic example, but the purpose here is to demonstrate how to work with more than one window. In the same Python script as the LoginWindow class, create a new class, MainWindow, that inherits QWidget. This new class can be seen in Listing 3-22.

Listing 3-22. The MainWindow class

```
# login_gui.py
class MainWindow(QWidget):

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

 def initializeUI(self):
 """Set up the application's GUI."""
 self.setMinimumSize(640, 426)
 self.setWindowTitle('3.1 – Main Window')
```

```
 self.setUpMainWindow()

 def setUpMainWindow(self):
 """Create and arrange widgets in the main window."""  image =
"images/background_kingfisher.jpg"

 try:
 with open(image):
 main_label = QLabel(self)
 pixmap = QPixmap(image)
 main_label.setPixmap(pixmap)
 main_label.move(0, 0)
 except FileNotFoundError as error:
 print(f"Image not found.\nError: {error}")
```

The size of the window is set using setMinimumSize(). One thing to note is that there is no show() method called in initializeUI(). This is because the window will only appear after a successful login (seen in Listing 3-18). The main window in Figure 3-10 presents a simple window with a QLabel.

The classes you create can inherit a majority of PyQt's classes, including ones for dialogs, as you shall see in the next section.

Explanation for Creating the Registration Dialog

Customization is one of the greatest benefits of using PyQt to build GUIs. When the user wants to register a new user, the dialog in Figure 3-9 appears. For the Registration dialog in Listings 3-23 to 3-27, create a separate Python script to keep the code organized and to demonstrate how to import your own custom classes into your projects.

Listing 3-23. Code for setting up the Registration dialog that inherits QDialog

```
# registration.py
# Import necessary modules
import sys
from PyQt6.QtWidgets import (QApplication, QDialog, QLabel,
QPushButton, QLineEdit, QMessageBox)
from PyQt6.QtGui import QFont, QPixmap
```

```
self.setModal(True)  self.initializeUI()

class NewUserDialog(QDialog):
                                         def initializeUI(self):
```

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

 """Set up the application's GUI."""
 self.setFixedSize(360, 320)
 self.setWindowTitle("3.1 - Registration GUI")
 self.setUpWindow()
```

The Registration dialog contains two QLabel widgets for the header and user image in Figure 3-10. These are created in Listing 3-24.

Listing 3-24. Adding labels in the Registration dialog's setUpWindow() method

```
# registration.py
 def setUpWindow(self):
 """Create and arrange widgets in the window for
 collecting new account information."""
 login_label = QLabel("Create New Account", self)
login_label.setFont(QFont("Arial", 20))
 login_label.move(90, 20)

 # Create QLabel for image
 user_image = "images/new_user_icon.png"

 try:
 with open(user_image):
 user_label = QLabel(self)
 pixmap = QPixmap(user_image)
 user_label.setPixmap(pixmap)
 user_label.move(150, 60)
 except FileNotFoundError as error:
 print(f"Image not found. Error: {error}")
```

Next, four QLineEdit widgets and their corresponding labels are created in Listing 3-25.

Listing 3-25. Adding labels and line editing widgets in the Registration dialog's setUpWindow() method

```
# registration.py
 # Create name QLabel and QLineEdit widgets
 name_label = QLabel("Username:", self)
 name_label.move(20, 144)

 self.name_edit = QLineEdit(self)
 self.name_edit.resize(250, 24)
 self.name_edit.move(90, 140)

 full_name_label = QLabel("Full Name:", self)
 full_name_label.move(20, 174)

 full_name_edit = QLineEdit(self)
 full_name_edit.resize(250, 24)
 full_name_edit.move(90, 170)

 # Create password QLabel and QLineEdit widgets
```

```
new_pswd_label = QLabel("Password:", self)
new_pswd_label.move(20, 204)

self.new_pswd_edit = QLineEdit(self)
self.new_pswd_edit.setEchoMode(
QLineEdit.EchoMode.Password)
self.new_pswd_edit.resize(250, 24)
self.new_pswd_edit.move(90, 200)

confirm_label = QLabel("Confirm:", self)
confirm_label.move(20, 234)

self.confirm_edit = QLineEdit(self)
self.confirm_edit.setEchoMode(
QLineEdit.EchoMode.Password)
self.confirm_edit.resize(250, 24)
self.confirm_edit.move(90, 230)
```

These widgets are used for collecting the user's username, full name, password, and an extra QLineEdit widget for ensuring that the password is entered correctly. The button for confirming the data is set up in Listing 3-26.

Listing 3-26. Adding a signup button in the Registration dialog's setUpWindow() method

```
# registration.py
 # Create sign up QPushButton
 sign_up_button = QPushButton("Sign Up", self)
 sign_up_button.resize(320, 32)
 sign_up_button.move(20, 270)
 sign_up_button.clicked.connect(self.confirmSignUp)
```

The sign_up_button emits a signal when clicked that calls the confirmSignUp() slot in Listing 3-27.

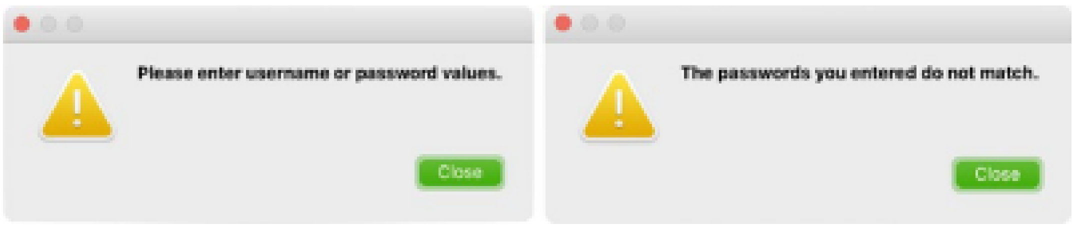
Tip QDialog has its own standard buttons that can be added to a custom dialog class using the QDialogButtonBox class.

Listing 3-27. Code for the confirmSignUp() slot

```
# registration.py
 def confirmSignUp(self):
 """Check if user information is entered and correct.  If so, append
username and password text to file."""     name_text =
self.name_edit.text()
 pswd_text = self.new_pswd_edit.text()
 confirm_text = self.confirm_edit.text()

 if name_text == "" or pswd_text == "":
 # Display QMessageBox if passwords don't match
QMessageBox.warning(self, "Error Message",
 "Please enter username or password values.",
QMessageBox.StandardButton.Close,
 QMessageBox.StandardButton.Close)
```

```
 elif pswd_text != confirm_text:
 # Display QMessageBox if passwords don't match
QMessageBox.warning(self, "Error Message",
 "The passwords you entered do not match.",
QMessageBox.StandardButton.Close,
 QMessageBox.StandardButton.Close)
 else:
 # Return to login window if passwords match
 with open("files/users.txt", 'a+') as f:
 f.write("\n" + name_text + " ")
 f.write(pswd_text)
 self.close()
```

The confirmSignUp() slot first reads the text from name_edit, new_pswd_edit, and confirm_edit. Next, a series of checks are performed to test if name_text or pswd_text is empty and then to see if pswd_text and confirm_text are the same. One of the two QMessageBox dialogs in Figure 3-13 will pop up if either of the conditions are met.


Please enter username or password values.

The passwords you entered do not match.

Figure 3-13. *A warning dialog that handles if the username or password fields are empty (left) and another that lets the user know the passwords don't match (right)*

If the user clicks the sign_up_button and all of the data is entered correctly, name_ text and pswd_text are saved on a new line in the users.txt file separated by a space that can be seen in Figure 3-14. If the user closes the dialog before completing the form, the dialog will close, but the Login window will still remain open.

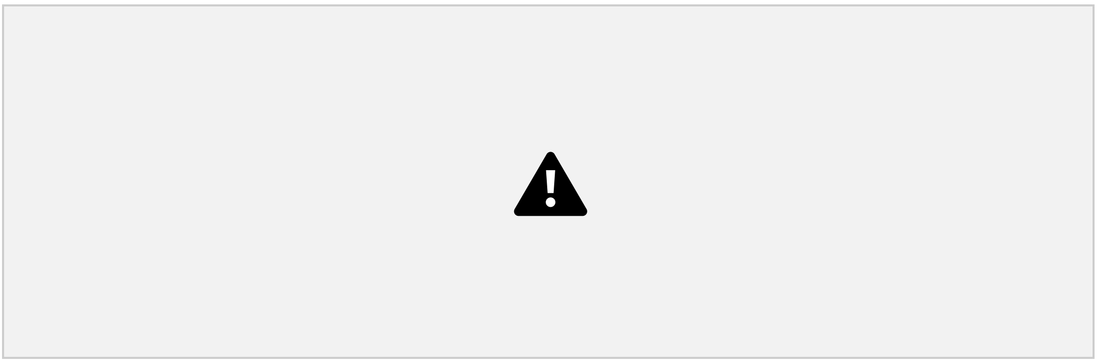


Figure 3-14. *The original users.txt file (left) and the updated one (right)*

If the form was completed, the user can try to enter their new username and password into the Login GUI to log in.

Summary

In this chapter, you experienced building GUIs using a variety of widgets – QPushButton, QLineEdit, QCheckBox, and QMessageBox. With the different widgets, you were also able to learn about other important concepts for building GUIs in PyQt, namely, event handling, communication between widgets with signals and slots, the difference between windows and dialogs, and how to create applications with multiple windows. All of these concepts in this chapter lay the framework for creating larger, more responsive, GUIs.

However, there are still a few more concepts that are essential to learn for creating GUI applications. In the next chapter, you will learn about another one of those fundamental topics – layout management.