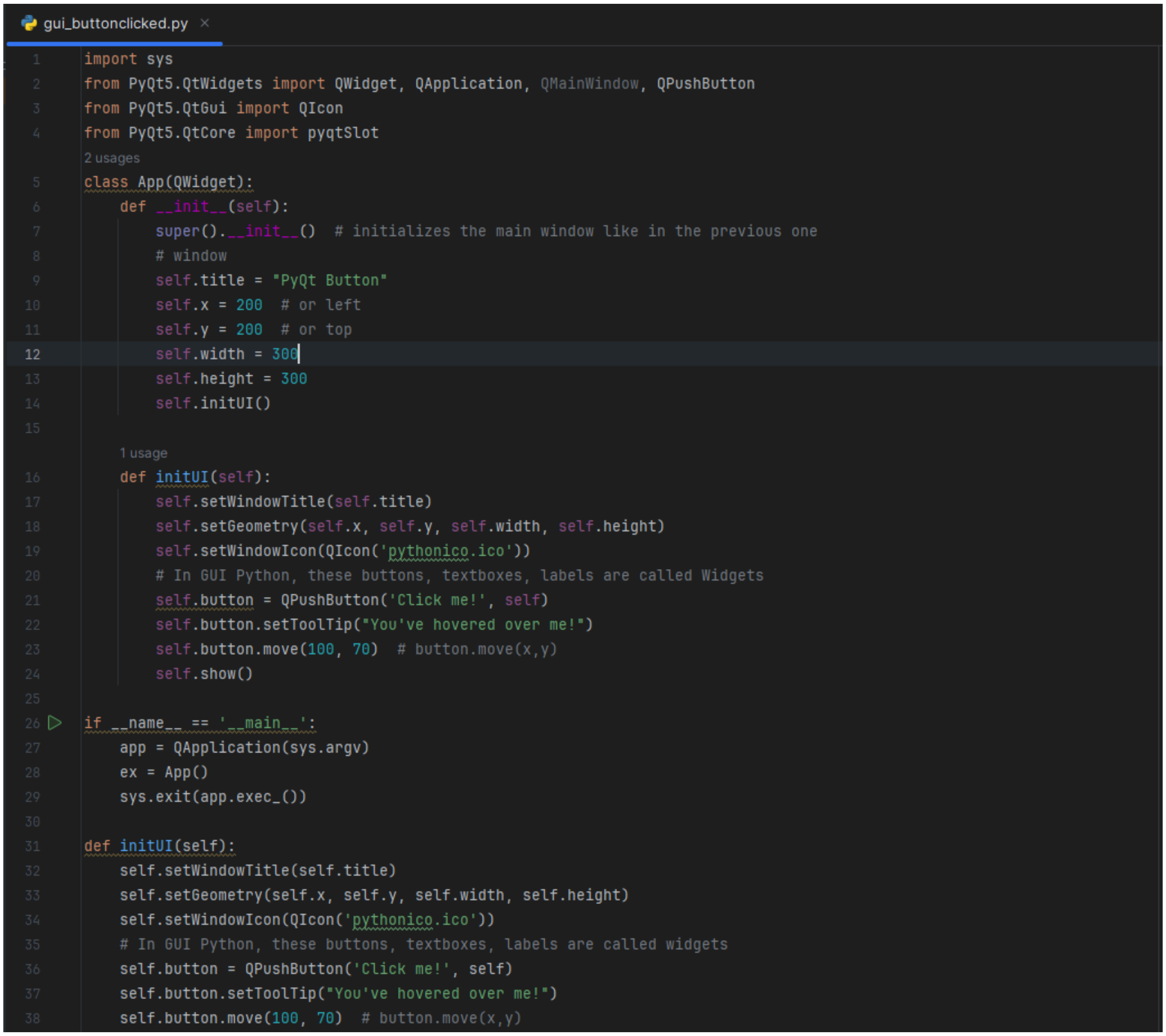
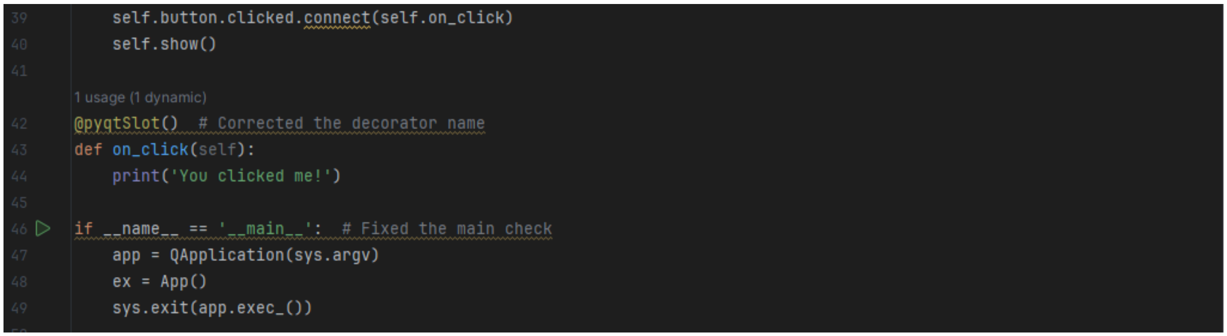
Laboratory Activity 5 - Introduction to Event Handling in GUI Development	
Rio, Aries, C.	10/21/24
BSCPE - CPE21S4	Ma’am MAria Rizette Sayo

Event Handling

Code:

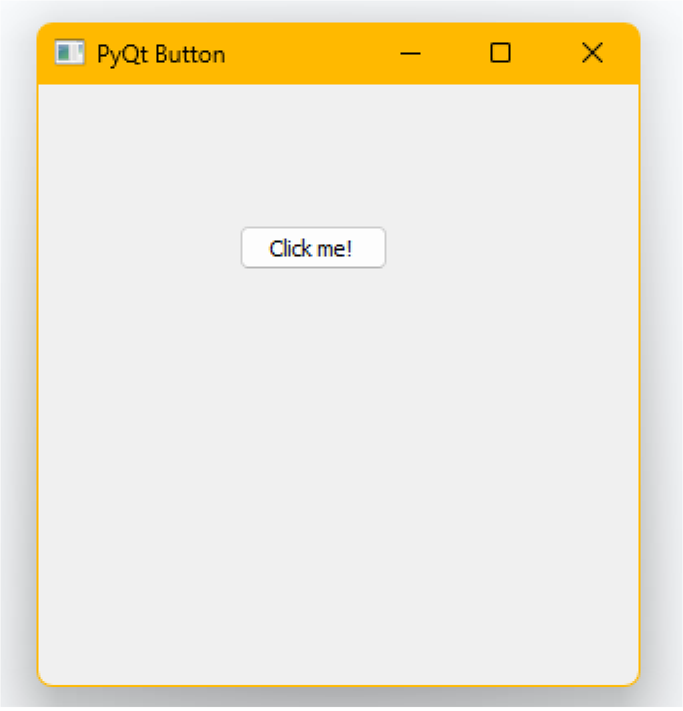


```python
import sys
from PyQt5.QtWidgets import QWidget, QApplication, QMainWindow, QPushButton
from PyQt5.QtGui import QIcon
from PyQt5.QtCore import pyqtSlot

class App(QWidget):
    def __init__(self):
        super().__init__()  # initializes the main window like in the previous one
        # window
        self.title = "PyQt Button"
        self.x = 200  # or left
        self.y = 200  # or top
        self.width = 300
        self.height = 300
        self.initUI()


    def initUI(self):
        self.setWindowTitle(self.title)
        self.setGeometry(self.x, self.y, self.width, self.height)
        self.setWindowIcon(QIcon('pythonico.ico'))
        # In GUI Python, these buttons, textboxes, labels are called Widgets
        self.button = QPushButton('Click me!', self)
        self.button.setToolTip("You've hovered over me!")
        self.button.move(100, 70)  # button.move(x,y)
        self.show()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = App()
    sys.exit(app.exec_())

def initUI(self):
    self.setWindowTitle(self.title)
    self.setGeometry(self.x, self.y, self.width, self.height)
    self.setWindowIcon(QIcon('pythonico.ico'))
    # In GUI Python, these buttons, textboxes, labels are called widgets
    self.button = QPushButton('Click me!', self)
    self.button.setToolTip("You've hovered over me!")
    self.button.move(100, 70)  # button.move(x,y)
```



```python
    self.button.clicked.connect(self.on_click)
    self.show()


@pyqtSlot()  # Corrected the decorator name
def on_click(self):
    print('You clicked me!')

if __name__ == '__main__':  # Fixed the main check
    app = QApplication(sys.argv)
    ex = App()
    sys.exit(app.exec_())
```

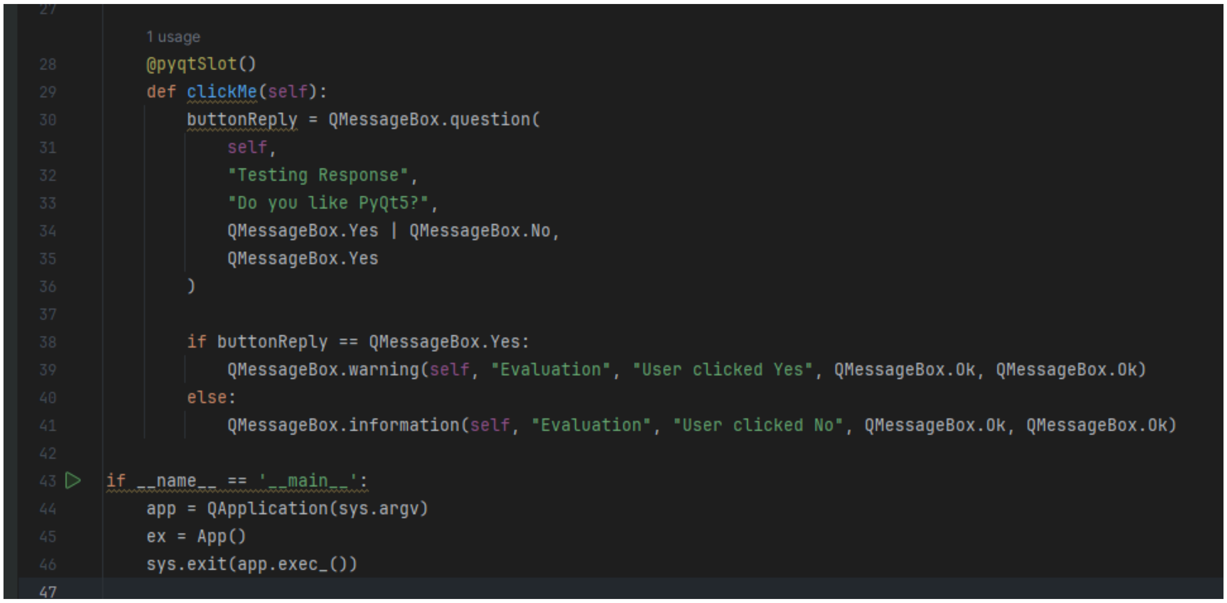
Output:



When you run the PyQt application and click the "Click me!" button, the terminal outputs "You clicked me!" indicating that the button click event is successfully connected to the `on_click` method. This demonstrates that the GUI is functioning properly, and the button is responding as expected to user interactions.

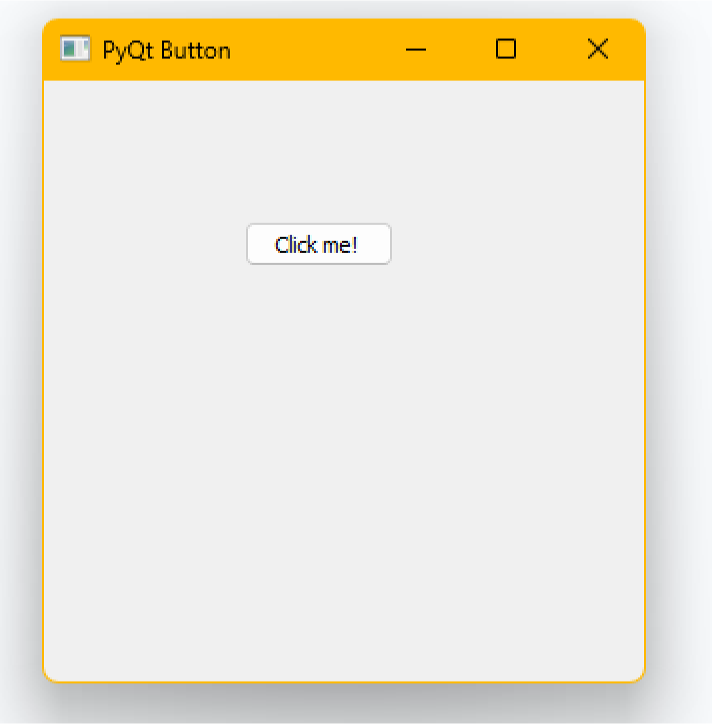
Adding a Message

Code:



```
    1 usage
    @pyqtSlot()
    def clickMe(self):
        buttonReply = QMessageBox.question(
            self,
            "Testing Response",
            "Do you like PyQt5?",
            QMessageBox.Yes | QMessageBox.No,
            QMessageBox.Yes
        )

        if buttonReply == QMessageBox.Yes:
            QMessageBox.warning(self, "Evaluation", "User clicked Yes", QMessageBox.Ok, QMessageBox.Ok)
        else:
            QMessageBox.information(self, "Evaluation", "User clicked No", QMessageBox.Ok, QMessageBox.Ok)

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

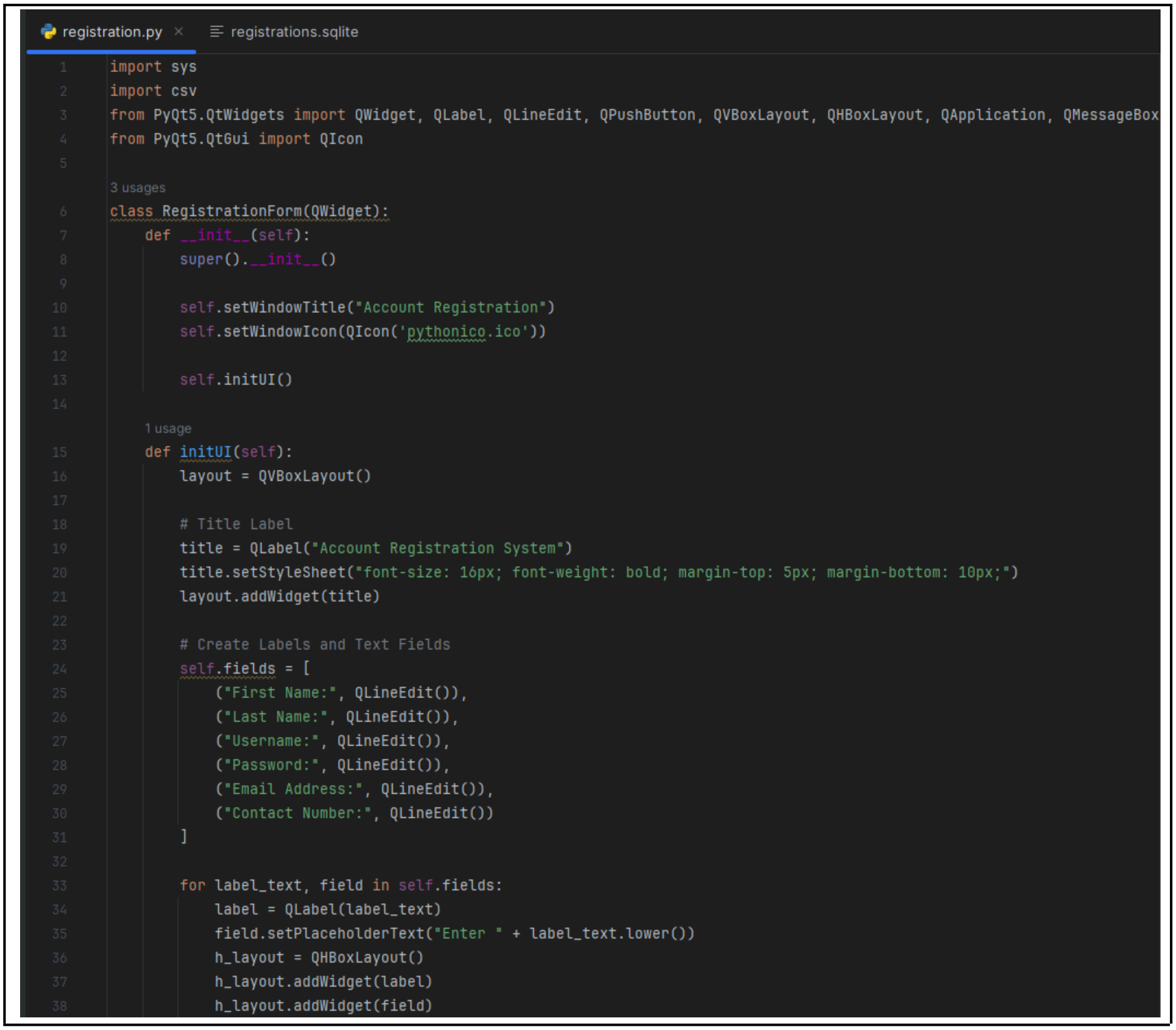
Output:



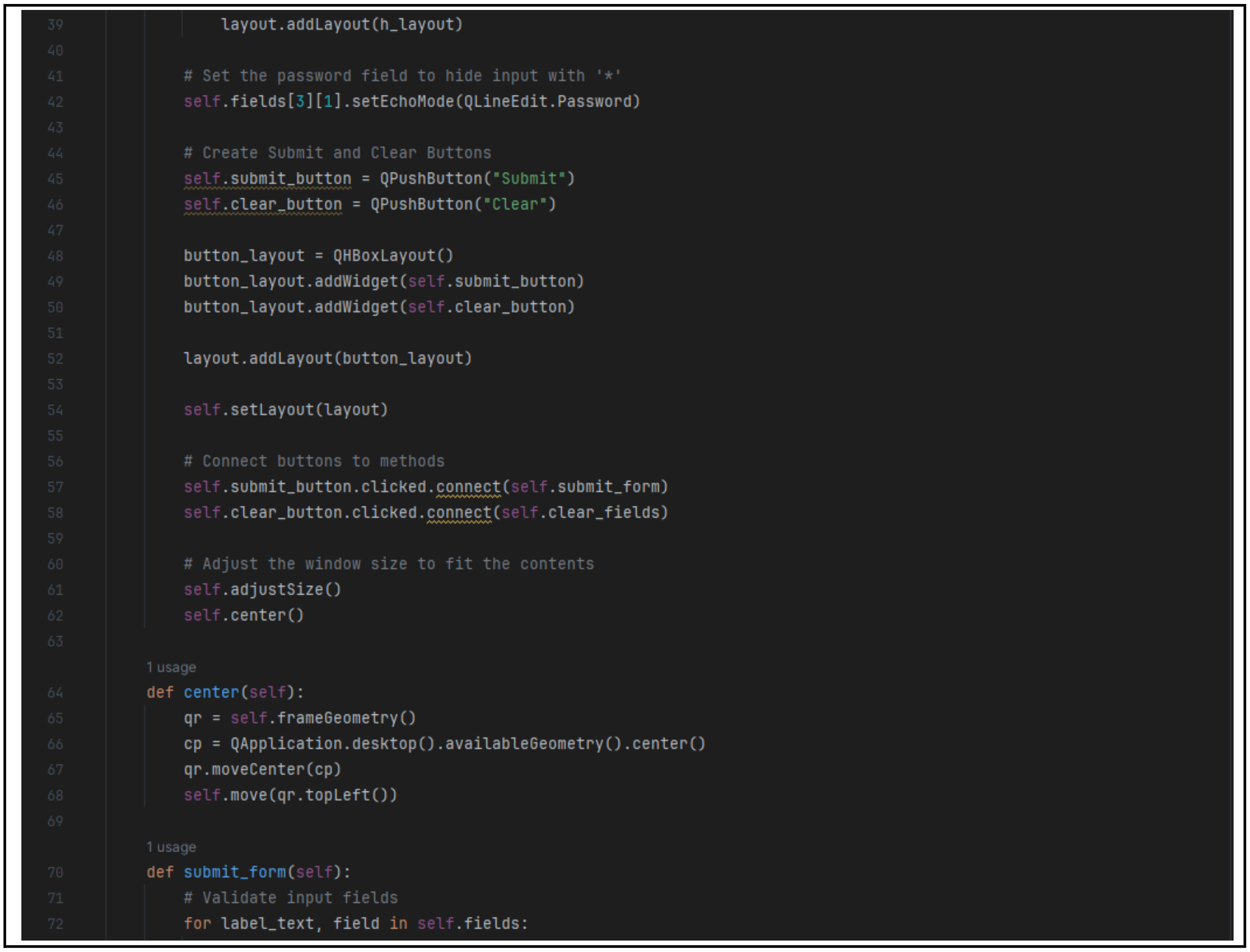
When you run the modified PyQt application and click the "Click me!" button, a message box appears asking, "Do you like PyQt5?" The user can respond with "Yes" or "No." If "Yes" is selected, a warning message box will inform that the user clicked "Yes." If "No" is chosen, an information message box will indicate that the user clicked "No." This interaction demonstrates how to handle user input and display responses using message boxes in a PyQt application.

Supplementary Activity

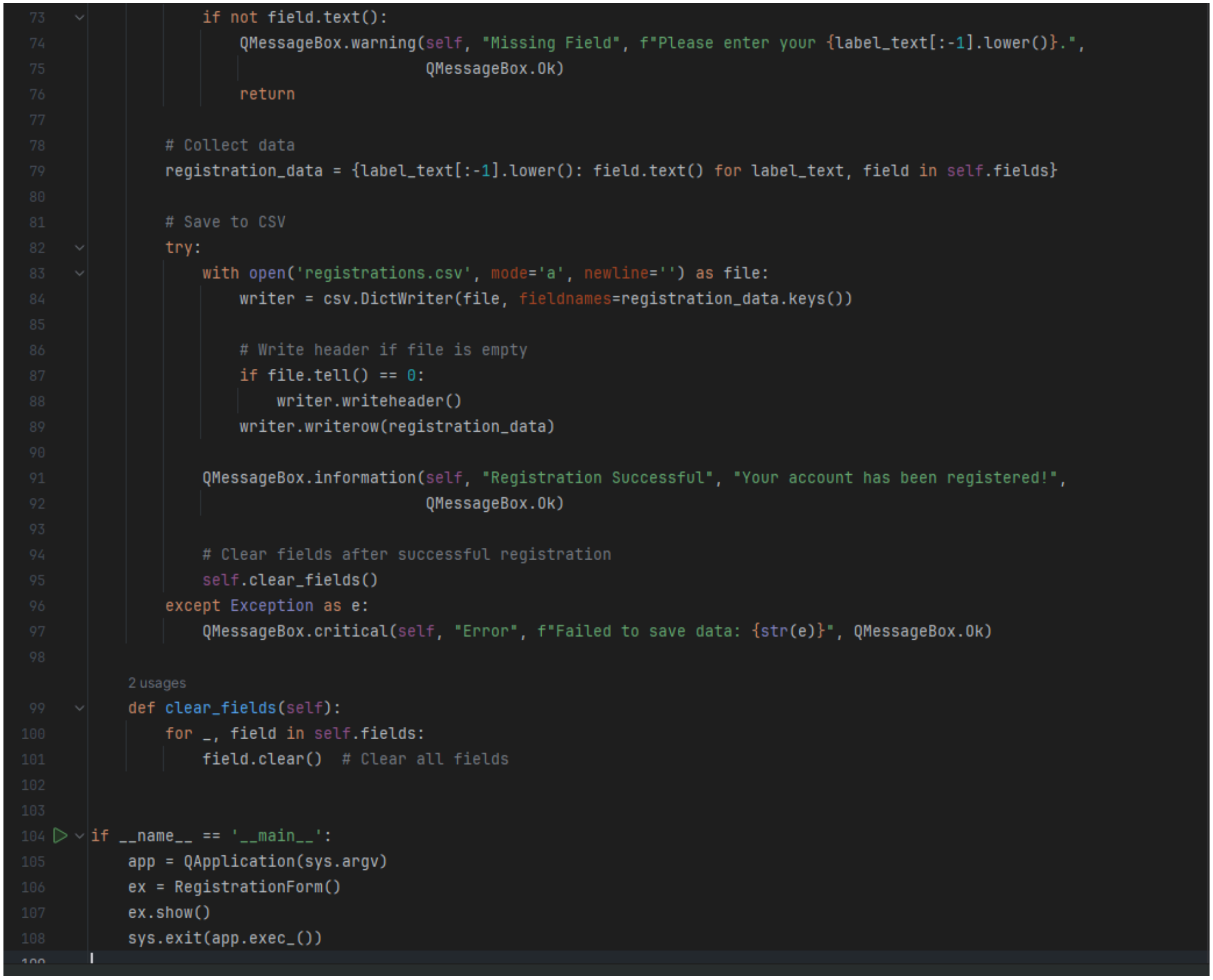
Code:



```python
import sys
import csv
from PyQt5.QtWidgets import QWidget, QLabel, QLineEdit, QPushButton, QVBoxLayout, QHBoxLayout, QApplication, QMessageBox
from PyQt5.QtGui import QIcon

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

        self.setWindowTitle("Account Registration")
        self.setWindowIcon(QIcon('pythonico.ico'))

        self.initUI()

    def initUI(self):
        layout = QVBoxLayout()

        # Title Label
        title = QLabel("Account Registration System")
        title.setStyleSheet("font-size: 16px; font-weight: bold; margin-top: 5px; margin-bottom: 10px;")
        layout.addWidget(title)

        # Create Labels and Text Fields
        self.fields = [
            ("First Name:", QLineEdit()),
            ("Last Name:", QLineEdit()),
            ("Username:", QLineEdit()),
            ("Password:", QLineEdit()),
            ("Email Address:", QLineEdit()),
            ("Contact Number:", QLineEdit())
        ]

        for label_text, field in self.fields:
            label = QLabel(label_text)
            field.setPlaceholderText("Enter " + label_text.lower())
            h_layout = QHBoxLayout()
            h_layout.addWidget(label)
            h_layout.addWidget(field)
```



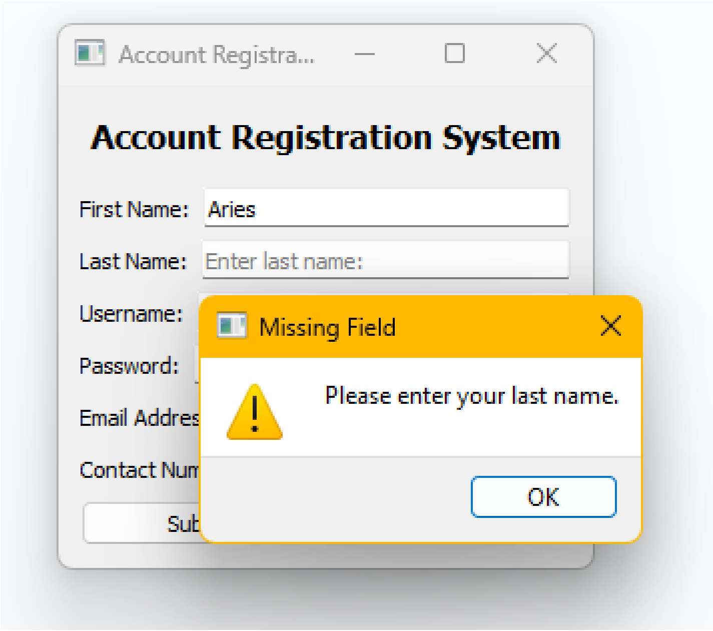
```
            layout.addLayout(h_layout)

        # Set the password field to hide input with '*'
        self.fields[3][1].setEchoMode(QLineEdit.Password)

        # Create Submit and Clear Buttons
        self.submit_button = QPushButton("Submit")
        self.clear_button = QPushButton("Clear")

        button_layout = QHBoxLayout()
        button_layout.addWidget(self.submit_button)
        button_layout.addWidget(self.clear_button)

        layout.addLayout(button_layout)

        self.setLayout(layout)

        # Connect buttons to methods
        self.submit_button.clicked.connect(self.submit_form)
        self.clear_button.clicked.connect(self.clear_fields)

        # Adjust the window size to fit the contents
        self.adjustSize()
        self.center()

    def center(self):
        qr = self.frameGeometry()
        cp = QApplication.desktop().availableGeometry().center()
        qr.moveCenter(cp)
        self.move(qr.topLeft())

    def submit_form(self):
        # Validate input fields
        for label_text, field in self.fields:
```



```python
            if not field.text():
                QMessageBox.warning(self, "Missing Field", f"Please enter your {label_text[:-1].lower()}.",
                                    QMessageBox.Ok)
                return

        # Collect data
        registration_data = {label_text[:-1].lower(): field.text() for label_text, field in self.fields}

        # Save to CSV
        try:
            with open('registrations.csv', mode='a', newline='') as file:
                writer = csv.DictWriter(file, fieldnames=registration_data.keys())

                # Write header if file is empty
                if file.tell() == 0:
                    writer.writeheader()
                writer.writerow(registration_data)

            QMessageBox.information(self, "Registration Successful", "Your account has been registered!",
                                    QMessageBox.Ok)

            # Clear fields after successful registration
            self.clear_fields()
        except Exception as e:
            QMessageBox.critical(self, "Error", f"Failed to save data: {str(e)}", QMessageBox.Ok)

    def clear_fields(self):
        for _, field in self.fields:
            field.clear()  # Clear all fields


if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = RegistrationForm()
    ex.show()
    sys.exit(app.exec_())
```

Output:

- Missing Field Notification



Account Registra...

Account Registration System

First Name: Aries

Last Name: Enter last name:

Username:

Password:

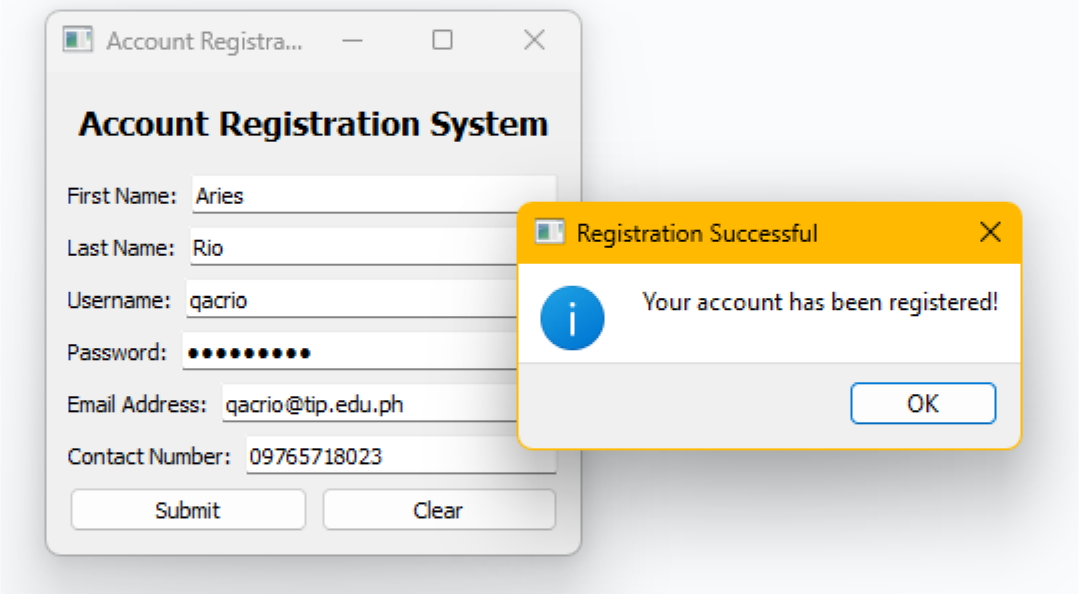
Email Addres

Contact Num

Missing Field

Please enter your last name.

- Successful Registration Confirmation



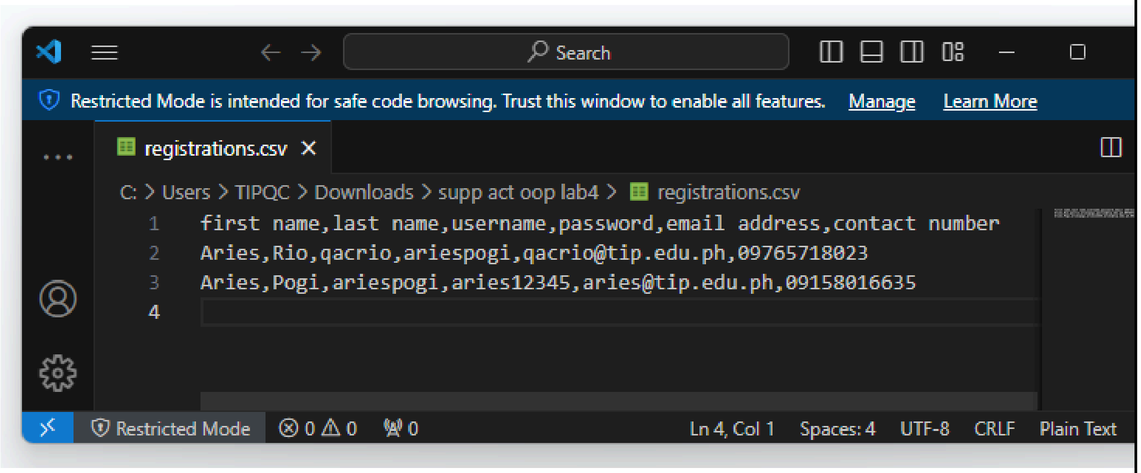
Account Registration System

First Name: Aries
Last Name: Rio
Username: qacrio
Password: ●●●●●●●●●
Email Address: qacrio@tip.edu.ph
Contact Number: 09765718023

Registration Successful

Your account has been registered!

- Saved Database


Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features.
Manage
Learn More
registrations.csv
C: > Users > TIPQC > Downloads > supp act oop lab4 > registrations.csv
Restricted Mode
0 0
0
Ln 4, Col 1
Spaces: 4
UTF-8
CRLF
Plain Text

```
first name,last name,username,password,email address,contact number
Aries,Rio,qacrio,ariespogi,qacrio@tip.edu.ph,09765718023
Aries,Pogi,ariespogi,aries12345,aries@tip.edu.ph,09158016635

```

Questions

1. What are the other signals available in PyQt5? (give at least 3 and describe each)

   PyQt5 provides various signals that facilitate interaction within GUI applications. One notable signal is the `clicked` signal, which is emitted when a button is clicked, enabling developers to execute specific actions in response. Another signal is `textChanged`, emitted whenever the text in a `QLineEdit` changes, useful for implementing real-time validation or updates. Lastly, the `currentIndexChanged` signal is emitted when the current selection in a `QComboBox` changes, allowing the application to respond dynamically to user selections.

2. Why do you think that event handling in Python is divided into signals and slots?

Event handling in Python is divided into signals and slots to create a clear and organized mechanism for managing user interactions. Signals represent events that occur, while slots are the functions that handle these events. This separation enhances modularity, allowing different components of an application to communicate without being tightly coupled, which improves code readability and maintainability. It also provides flexibility for developers to connect and disconnect functions easily as needed.

3. How can message boxes be used to provide a better User Experience or how can message boxes be used to make a GUI Application more user-friendly?

   Message boxes can significantly enhance user experience by providing immediate feedback and guidance in a clear format. They can alert users to errors, confirm important actions, or convey critical information, helping users navigate the application effectively. By utilizing well-designed message boxes, developers can clarify user actions, reducing confusion and improving the overall usability of the application.

4. What is Error-handling and how was it applied in the task performed?

   Error-handling is the process of anticipating, detecting, and responding to errors that may occur during a program's execution. In the task performed, error-handling was applied through the use of `try` and `except` blocks to manage exceptions when saving registration data to a CSV file. If an error occurs, a message box informs the user, preventing application crashes and providing a graceful response to unexpected situations.

5. What may be the reasons behind the need to implement error handling?

   Implementing error handling is crucial for building robust applications that can handle unexpected issues gracefully. It prevents application crashes, ensuring stability and maintaining user trust. Error handling also aids in debugging by capturing error details, allowing developers to identify and resolve problems efficiently. Furthermore, it enhances user experience by providing informative messages instead of leaving users confused during failures.

Conclusion

In conclusion, effective event handling, including the use of signals, slots, and message boxes, is essential for creating user-friendly GUI applications in PyQt5. Error handling is vital for ensuring reliability and improving user experience by managing unexpected issues effectively. By applying these concepts, developers can create intuitive and robust applications that meet user needs and expectations.