NAVTTEC

STUDENT MANAGEMENT DATABASE

PYQT5 PYTHON

Group Assignment
Seemal Naeem
Umm-ul-Baneen
Laiba
Isha Imran
Muniba Khalid

ABSTRACT

The Student Management System project uses PyQt5 and Python modules to create a user-friendly software solution for school management. It includes features such as secure login, student information management, tracking of fees and attendance, and data permanence. The system has an easy-to-use interface and keeps student information up to date for effective management.

INTRODUCTION

Student Management System is a software which is helpful for school management. In this current system, all the activities are done manually. In this software, management can log in as a user and can add, delete, and update students. Also, there is an option to manage the fees and attendance of students. This proposed system has several advantages including a user-friendly interface, fast access to a database, look and feel environment.

METHODOLOGY

In this project, PyQt5, a GUI module that connects the Qt C++ cross-platform framework with the Python language is used. The Qt class's function helps in communication between items to design reusable software components with ease. Also, Qt comes with Qt Designer, a tool that acts as a graphical user interface. PyQt5 can design Python code from Qt Designer while adding new GUI controls when both Qt Designer and Python programming languages are used.

SOFTWARES

The software used in this project includes:

- PyCharm
- Qt Designer

LIBRARIES

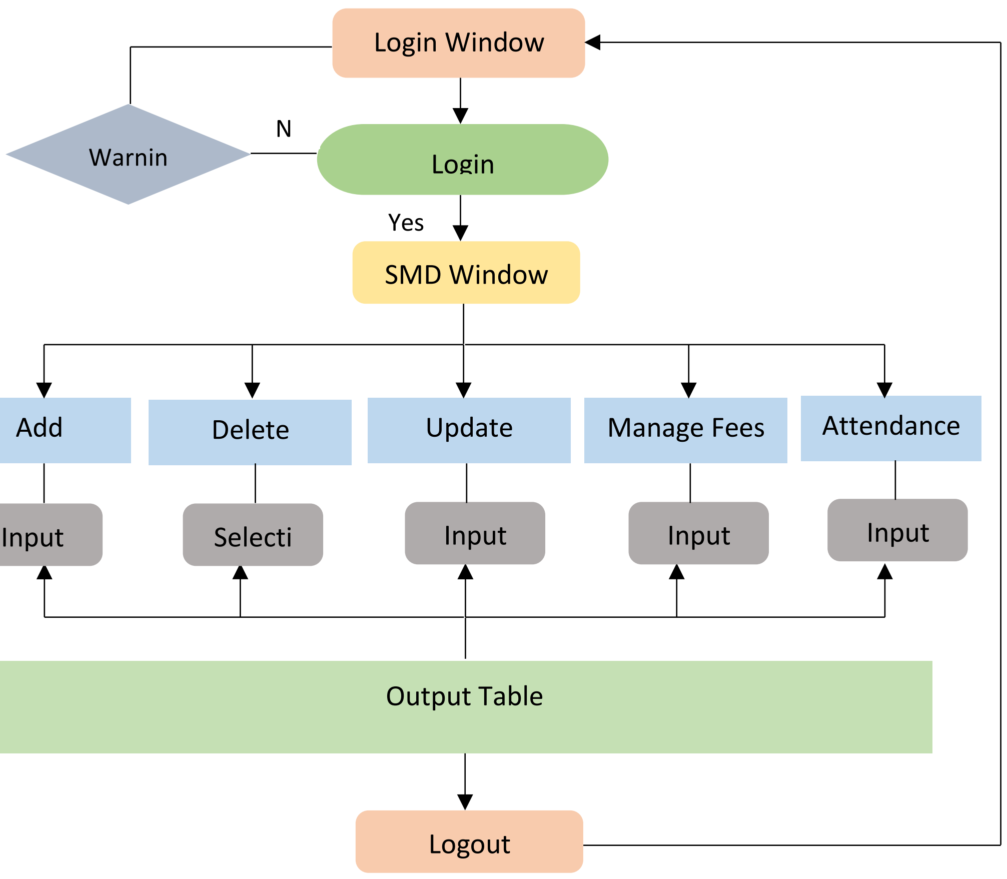
In this project, different Python libraries are used including:

1- **sys**: This library provides access to some variables used or maintained by the interpreter and to functions that interact with the interpreter.
2- **CSV**: This library provides functionality for reading and writing CSV (Comma Separated Values) files.
3- **PyQt5**: This library provides a set of Python bindings for Qt, a powerful cross-platform application framework. It is used for creating graphical user interfaces (GUIs).

MAIN FUNCTIONS

- setupUi: Sets up the UI layout, and styles, and connects the login button to the openwindow1 function.
- retranslateUi: Translates text on UI elements.
- __init__ (MainWindow constructor): Initializes the main window, creates widgets, and connects button clicks to functions.

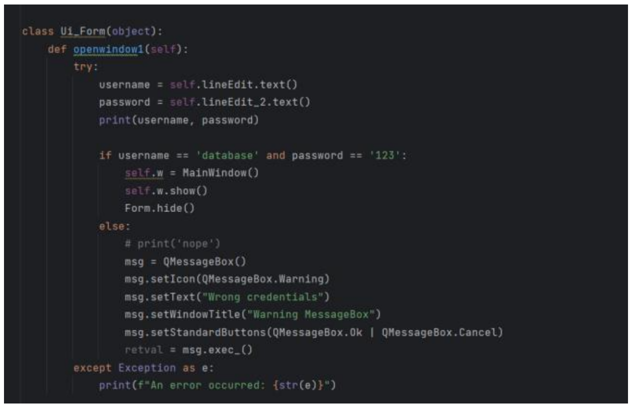
FLOW DIAGRAM:



FUNCTIONS:

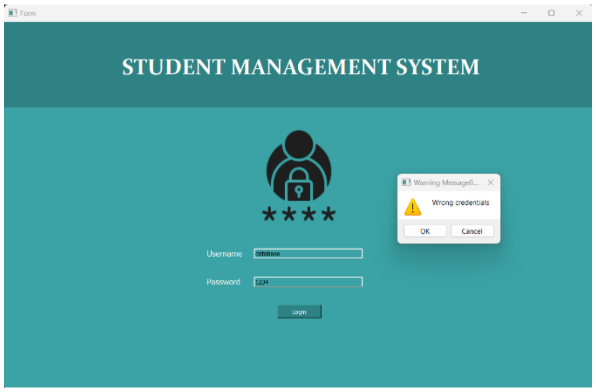
- Login

In login functionality, code captures the values entered in two line edit widgets (username and password), checks if they match a specific username and password, and based on the result, either open a new window or displays a warning message using a QMessageBox.



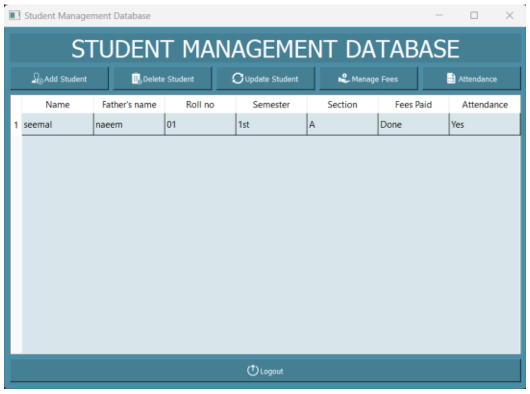
```python
class Ui_Form(object):
    def openwindow1(self):
        try:
            username = self.lineEdit.text()
            password = self.lineEdit_2.text()
            print(username, password)

            if username == 'database' and password == '123':
                self.w = MainWindow()
                self.w.show()
                Form.hide()
            else:
                # print('nope')
                msg = QMessageBox()
                msg.setIcon(QMessageBox.Warning)
                msg.setText("Wrong credentials")
                msg.setWindowTitle("Warning MessageBox")
                msg.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
                retval = msg.exec_()
        except Exception as e:
            print(f"An error occurred: {str(e)}")
```



Wrong credentials

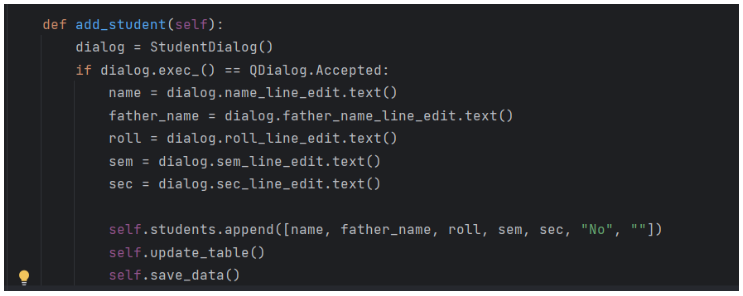
Username: database
Password: 1234



	Name	Father's name	Roll no	Semester	Section	Fees Paid	Attendance
1	seemal	naeem	01	1st	A	Done	Yes

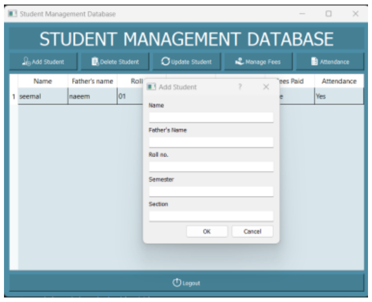
- Add

This code piece is used to add a new student's information to a list (self.students) and update a table widget to reflect the new data. It assumes that a custom dialog window (StudentDialog) is used to capture the student's information. The save_data() method is responsible for persisting the data, but its implementation is not provided here.



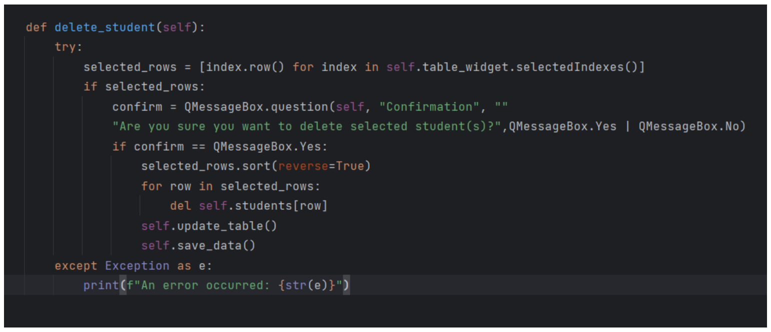
```
def add_student(self):
    dialog = StudentDialog()
    if dialog.exec_() == QDialog.Accepted:
        name = dialog.name_line_edit.text()
        father_name = dialog.father_name_line_edit.text()
        roll = dialog.roll_line_edit.text()
        sem = dialog.sem_line_edit.text()
        sec = dialog.sec_line_edit.text()

        self.students.append([name, father_name, roll, sem, sec, "No", ""])
        self.update_table()
        self.save_data()
```

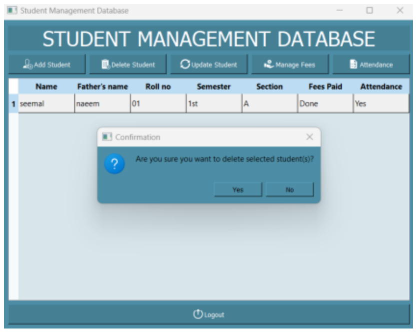


- Delete

The delete_student function deletes the selected student(s) from the self.students list based on the selected rows in the table_widget. It asks for confirmation from the user before performing the deletion. After deleting the selected student(s), it updates the table widget and saves the updated data. If any exceptions occur, they are caught and an error message is printed.



```
def delete_student(self):
    try:
        selected_rows = [index.row() for index in self.table_widget.selectedIndexes()]
        if selected_rows:
            confirm = QMessageBox.question(self, "Confirmation", ""
            "Are you sure you want to delete selected student(s)?",QMessageBox.Yes | QMessageBox.No)
            if confirm == QMessageBox.Yes:
                selected_rows.sort(reverse=True)
                for row in selected_rows:
                    del self.students[row]
                self.update_table()
                self.save_data()
    except Exception as e:
        print(f"An error occurred: {str(e)}")
```

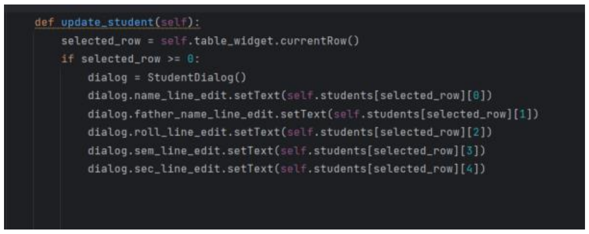


	Name	Father's name	Roll no	Semester	Section	Fees Paid	Attendance
1	seemal	naeem	01	1st	A	Done	Yes

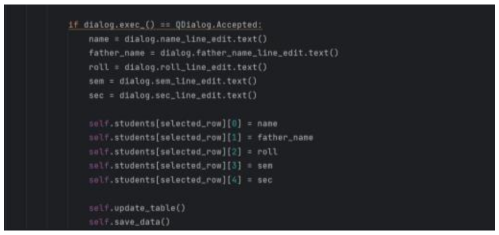
Are you sure you want to delete selected student(s)?

- Update

This code snippet is used to update a selected student's information in the students list depending on modifications made in a custom dialogue window (StudentDialog). The update_table() function is in charge of updating the table widget, and the save_data() method is in charge of storing the modified data, however, their implementations are not included here.



```python
def update_student(self):
    selected_row = self.table_widget.currentRow()
    if selected_row >= 0:
        dialog = StudentDialog()
        dialog.name_line_edit.setText(self.students[selected_row][0])
        dialog.father_name_line_edit.setText(self.students[selected_row][1])
        dialog.roll_line_edit.setText(self.students[selected_row][2])
        dialog.sem_line_edit.setText(self.students[selected_row][3])
        dialog.sec_line_edit.setText(self.students[selected_row][4])
```



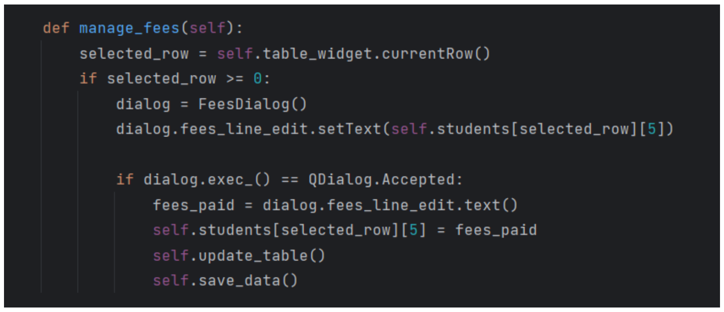
```python
        if dialog.exec_() == QDialog.Accepted:
            name = dialog.name_line_edit.text()
            father_name = dialog.father_name_line_edit.text()
            roll = dialog.roll_line_edit.text()
            sem = dialog.sem_line_edit.text()
            sec = dialog.sec_line_edit.text()

            self.students[selected_row][0] = name
            self.students[selected_row][1] = father_name
            self.students[selected_row][2] = roll
            self.students[selected_row][3] = sem
            self.students[selected_row][4] = sec

            self.update_table()
            self.save_data()
```

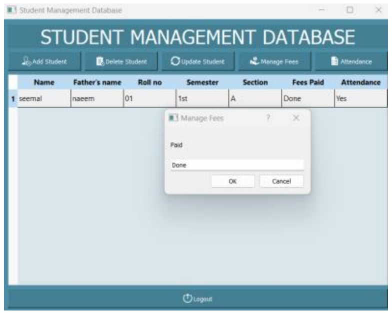
- Fees

This code snippet is used to handle a specific student's fees. It displays the current fee information in a custom dialogue window (FeesDialog) and allows the user to edit it. Following the acceptance of the modifications, the code updates the students list, and the table widget, and stores the updated data.



```python
def manage_fees(self):
    selected_row = self.table_widget.currentRow()
    if selected_row >= 0:
        dialog = FeesDialog()
        dialog.fees_line_edit.setText(self.students[selected_row][5])

        if dialog.exec_() == QDialog.Accepted:
            fees_paid = dialog.fees_line_edit.text()
            self.students[selected_row][5] = fees_paid
            self.update_table()
            self.save_data()
```



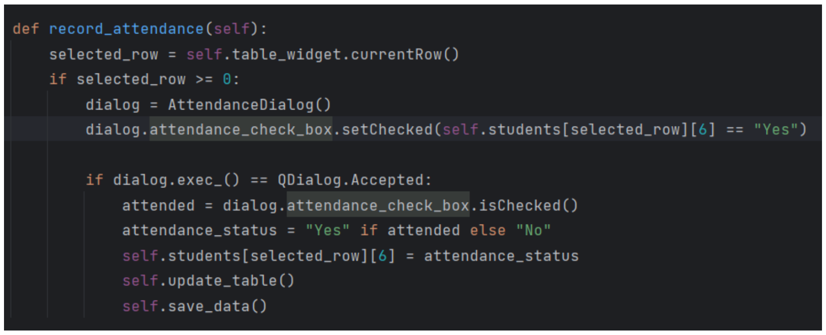
	Name	Father's name	Roll no	Semester	Section	Fees Paid	Attendance
1	seemal	naeem	01	1st	A	Done	Yes

Paid

Done

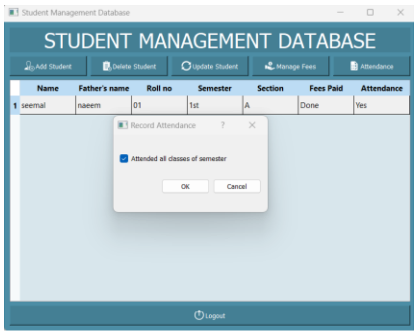
- Attendance

This code snippet is used to track a specific student's attendance. It displays the current attendance status in a custom dialogue window (AttendanceDialog) and allows the user to adjust it with a checkbox. Following the acceptance of the modifications, the code updates the students list, and the table widget, and stores the updated data.



```python
def record_attendance(self):
    selected_row = self.table_widget.currentRow()
    if selected_row >= 0:
        dialog = AttendanceDialog()
        dialog.attendance_check_box.setChecked(self.students[selected_row][6] == "Yes")

        if dialog.exec_() == QDialog.Accepted:
            attended = dialog.attendance_check_box.isChecked()
            attendance_status = "Yes" if attended else "No"
            self.students[selected_row][6] = attendance_status
            self.update_table()
            self.save_data()
```

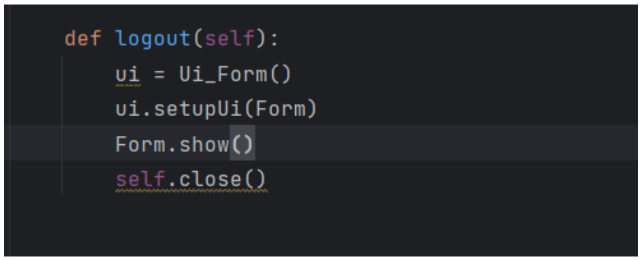


	Name	Father's name	Roll no	Semester	Section	Fees Paid	Attendance
1	seemal	naeem	01	1st	A	Done	Yes

✓ Attended all classes of semester

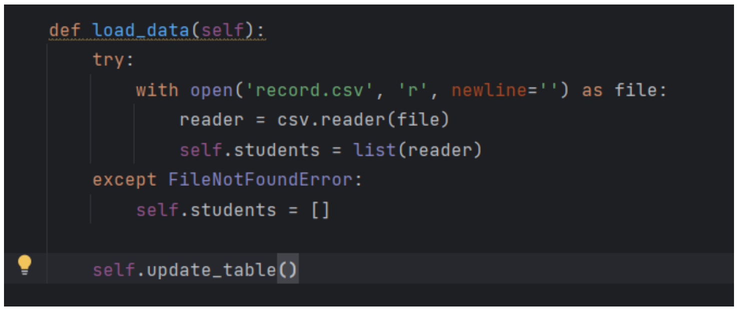
- Logout

The logout function is in charge of configuring and displaying the main form (after the user logs out), using the Ui_Form class and its setupUi method to configure the UI components before displaying the form.


```python
def logout(self):
    ui = Ui_Form()
    ui.setupUi(Form)
    Form.show()
    self.close()
```

- Load data

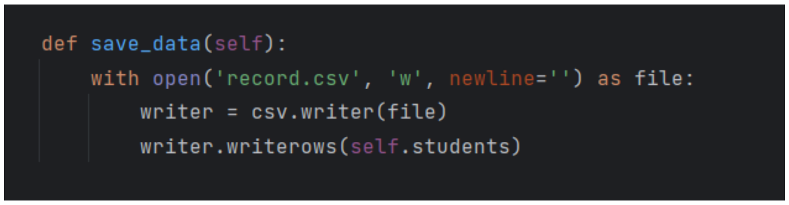
The load_data method tries to open and read the "record.csv" CSV file. If the file is located, the data is read using a CSV reader and stored in the students property. If the file cannot be retrieved, pupils are allocated to an empty list. Finally, the update_table function is used to update the GUI table with the supplied data.


```python
def load_data(self):
    try:
        with open('record.csv', 'r', newline='') as file:
            reader = csv.reader(file)
            self.students = list(reader)
    except FileNotFoundError:
        self.students = []

    self.update_table()
```

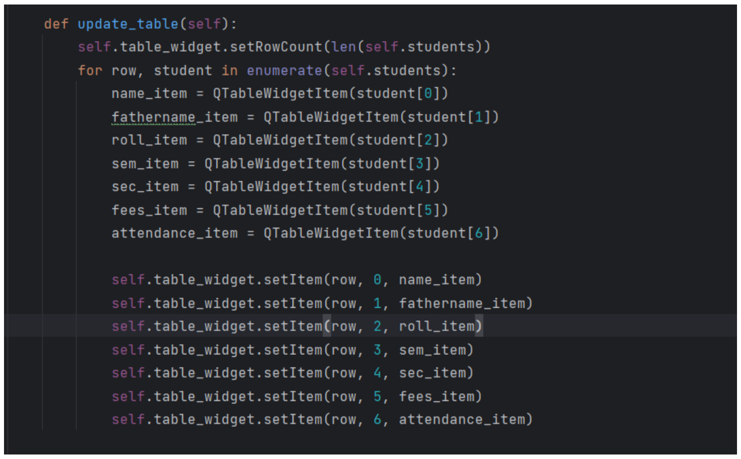
- Save data

The save_data function opens the "record.csv" file in write mode, creates a CSV writer object, and then writes the data from the self.students list to the file using the writerows method. This ensures that the data is saved in CSV format and persists for future use.


```python
def save_data(self):
    with open('record.csv', 'w', newline='') as file:
        writer = csv.writer(file)
        writer.writerows(self.students)
```

- Update data

The update_table function updates the table_widget to display the student records stored in the self.students list. It sets the number of rows in the table, creates QTableWidgetItem objects for each field of a student record, and sets these items in the appropriate cells of the table. This ensures that the table reflects the current state of the student data.



```
def update_table(self):
    self.table_widget.setRowCount(len(self.students))
    for row, student in enumerate(self.students):
        name_item = QTableWidgetItem(student[0])
        fathername_item = QTableWidgetItem(student[1])
        roll_item = QTableWidgetItem(student[2])
        sem_item = QTableWidgetItem(student[3])
        sec_item = QTableWidgetItem(student[4])
        fees_item = QTableWidgetItem(student[5])
        attendance_item = QTableWidgetItem(student[6])

        self.table_widget.setItem(row, 0, name_item)
        self.table_widget.setItem(row, 1, fathername_item)
        self.table_widget.setItem(row, 2, roll_item)
        self.table_widget.setItem(row, 3, sem_item)
        self.table_widget.setItem(row, 4, sec_item)
        self.table_widget.setItem(row, 5, fees_item)
        self.table_widget.setItem(row, 6, attendance_item)
```

CONCLUSION:

The objective of this software is to provide a framework that enables administrators to conveniently store the data of students. Several proposals are made to improve the functioning and usefulness of the Student Management System. Improving data validation, implementing search and filtering functionalities, enhancing reporting and analytics, implementing notifications and reminders, integrating with existing systems, improving the user interface, implementing data backup and recovery, providing user training and support, and gathering user feedback are some of these. These additions will provide effective data management and decision-making capabilities to school administrators, instructors, and staff.