

## Criterion C: Development

### Table of Contents

1. External Libraries Used.....	2
2. User-defined Modules.....	3
3. Database and SQL.....	3
4. Searching Algorithms and Methods.....	5
4.1. Searching for Locally Stored Files.....	5
4.2. Searching for Specific String in a File.....	5
5. Sorting Algorithm.....	6
6. Data Structures Used.....	7
6.1. Python Lists:.....	7
6.2. Python Dictionaries:.....	7
6.3. Python Tuples:.....	8
6.4. Python Sets:.....	8
7. Object Oriented Programming Features Used.....	8
7.1. Inheritance.....	8
7.2. Polymorphism.....	9
7.3. Classes with Static Methods.....	9
8. Editing the Spreadsheet.....	9
8.1. Inserting New Data Into New Spreadsheet.....	9
8.2. Inserting New Names Into Pre-existing Spreadsheet.....	9
8.3. Removing Names from Pre-existing Spreadsheet.....	13
8.4. Adding the Presence Status of Students into the Spreadsheet.....	14
9. Storing and Saving Records Locally.....	15
10. Recent Searches Feature Implementation.....	15
11. GUI Elements.....	17
12. Error Handling and Data Validation.....	21

## 1. External Libraries Used

A library is a collection of organized set of features (usually in the form of classes or functions). As, Python, as a programming language, natively does not have every library with every possible feature, it is necessary to program them manually. However, it is quite difficult and time consuming for a single programmer to program every feature. Hence, programmers make use of external libraries which are essentially collections of features programmed by someone other than the language vendor and the programmer themselves. These external libraries can be imported into any program and its features can be used. This Attendance Management Application, as also mentioned in the Planning Section, makes use, largely, of the Kivy Library. Some other external libraries and the specific classes within the Kivy library that are used in the program include:

```
# -----Standard Libraries-----
# from: https://docs.python.org/3/library/
import os
import shutil
import sys
import pickle
from datetime import datetime
import sqlite3
import hashlib

# -----Kivy-specific Imports-----
from kivy.app import App
from kivy.lang import Builder
from kivy.core.text import LabelBase
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.popup import Popup
from kivy.uix.button import Button
from kivy.core.window import Window

# -----Other External Libraries-----
import ntpath
import py7zr
import openpyxl
from plyer import filechooser
```

```
# -----
# Imports for .kv file
# -----

# -----Standard Libraries-----
# from: https://docs.python.org/3/library/
#: import webbrowser webbrowser

# -----Kivy-specific Imports-----
#: import FadeTransition kivy.uix.screenmanager.FadeTransition
#: import Factory kivy.factory.Factory
```

## 2. User-defined Modules

The program, as discussed in the Design Section, has been divided into various modules. These modules have been created so that the code is more readable because modules allow abstraction. The modules include: main.py, gui.kv, filemanager.py, attendance.py, users.py, database.py, help.html. Here, all the functions of the modules have been made the same as mentioned in the Design Section. However, the addition of help.html has been made to add a 'help' page to the application and has been written in HTML. This was done for the convenience of the user wanting assistance while using the software. The page is a form of user manual.

## 3. Database and SQL

The SQLite Database was used and was implemented in the program using the 'sqlite3' library. Some SQL queries used include:

1. **CREATE:** to create tables

```
class DbManager:
    @staticmethod
    def check_db():
        if not os.path.isfile('user_info.db'): # checking if the list returned is empty
            DbManager.init_user_db()
        return True

    @staticmethod
    def connect_user_db():
        conn = sqlite3.connect('user_info.db')
        c = conn.cursor()

        return conn, c

    @staticmethod
    def init_user_db():
        conn, c = DbManager.connect_user_db()

        c.execute('''CREATE TABLE users
(id INTEGER, username TEXT, password TEXT)''')

        conn.commit()
        conn.close()
```

## 2. **SELECT:** to retrieve data from the database

```
@staticmethod
def username_exists(username):
    conn, c = DbManager.connect_user_db()

    found = False
    for row in c.execute('SELECT username FROM users'):
        if row[0] == username:
            found = True

    conn.commit()
    conn.close()

    return found
```

```
@staticmethod
def get_pwd(username):
    conn, c = DbManager.connect_user_db()

    pwd = list(c.execute('SELECT password FROM users WHERE username = (?)', [(username)]))[0][0]

    conn.commit()
    conn.close()

    return pwd
```

## 3. **INSERT:** to add data to the database

```
@staticmethod
def add_user(username, password):
    conn, c = DbManager.connect_user_db()

    user_id = DbManager.get_id()
    hashed_password = DbManager.hash_password(password)

    values = [(user_id, username, hashed_password)]

    c.executemany('INSERT INTO users VALUES (?, ?, ?)', values)
    conn.commit()
    conn.close()
```

## 4. Searching Algorithms and Methods

### 4.1. Searching for Locally Stored Files

Searching for any files stored locally was done using the 'os' library. By using the 'os.walk' function, every directory and file was searched for using a linear-search-like algorithm to find the path of the specific required file. A user-defined function was created to achieve this. The 'specific' parameter affects whether to search for files that have a particular string from the search query in the file name or whether the file name is exactly same as the search query. 'specific' set to False is useful for the search feature (of the ViewRecords Window) in the software.

```
@staticmethod
def find_file(name_of_record, path='.', specific=False):
    result = []
    files_lower = []
    path = os.path.expanduser(path)
    if specific:
        for root, dirs, files in os.walk(path):
            files_lower[:] = [file.lower() for file in files]
            for index, file in enumerate(files_lower):
                if f'{name_of_record.lower()}_ta_app.xlsx' == file:
                    result.append(os.path.join(root, files[index]))
    else:
        for root, dirs, files in os.walk(path):
            files_lower[:] = [file.lower() for file in files]
            for index, file in enumerate(files_lower):
                if name_of_record.lower() in file and '_ta_app.xlsx' in file:
                    result.append(os.path.join(root, files[index]))
    return result
```

### 4.2. Searching for Specific String in a File

Using the Python 'in' operator, specific strings in a file are searched. This operators uses linear search which means that the string to be searched is compared with every string in the file until there is a match. This was used in searching for names of students in the Zoom Chat File.

```

class AttendanceHandler:
    def __init__(self, excel_sheet):
        self.excel_names = []
        self.excel_sheet = excel_sheet

    def get_present_students(self, chat_file):
        self.get_names_excel()
        present_students_tmp = []
        with open(chat_file) as chat_file:
            for line in chat_file:
                present_students_tmp.append([student for student in self.excel_names if student in line])
        present_students = [students for lists in present_students_tmp for students in lists]
        return present_students

    def get_names_excel(self):
        max_row = self.excel_sheet.max_row
        current_row = 2
        while current_row <= max_row:
            self.excel_names.append((self.excel_sheet.cell(row=current_row, column=2)).value)
            current_row += 1

```

## 5. Sorting Algorithm

The names of students stored in the spreadsheet needed to be sorted. For this, insertion sort algorithm was used because of its faster speed over bubble sort.

```

@staticmethod
def process_names(names):
    names = names.strip().split(',')
    for index, name in enumerate(names):
        names[index] = name.strip()
        if name == '':
            names.remove(name)
    names = list(set(names)) # remove repeating names

    # sorting names using insertion sort
    for i in range(1, len(names)):
        key = names[i]
        j = i - 1

        while j >= 0 and key < names[j]:
            names[j + 1] = names[j]
            j -= 1

        names[j + 1] = key

    return names

```

## 6. Data Structures Used

### 6.1. Python Lists:

Lists in python are dynamic, array-like data structures which allow duplicate values. This application makes extensive use of lists for various reasons some of which are mentioned in this Section. It is defined by enclosing the array of data within brackets [].

### 6.2. Python Dictionaries:

Dictionaries are dynamic data structures where data is stored in key:value pairs. Meaning all values stored in a dictionary are mapped to a unique key. The value can be of any other data type or structure such as a list. For the purpose of this application, one of the uses of dictionary was made when adding names to an already existing spreadsheet (this is explained further below in the Section Inserting Data Into Spreadsheet). A dictionary is defined by enclosing the key value pair in curly brackets as such {key1:value1, key2:value2}.

```
@staticmethod
def add_students_excel(names_of_students, name_of_record):
    additional_names = FileManager.process_names(names_of_students)

    excel_file = FileManager.find_file(name_of_record, path=f'users/{User.current_user}')[0]

    wb = openpyxl.load_workbook(excel_file)
    sheet = wb.active

    handler = AttendanceHandler(excel_file, sheet)
    handler.get_names_excel()

    current_names = handler.excel_names

    updated_record = {}

    max_column = sheet.max_column
    max_row = sheet.max_row

    for name in additional_names:
        updated_record[name] = [None]*(max_column - 2)

    current_row = 2
    while current_row <= max_row:
        name_of_student = (sheet.cell(row=current_row, column=2)).value
        current_column = 3
        presence = []
        while current_column <= max_column:
            presence.append((sheet.cell(row=current_row, column=current_column)).value)
            current_column += 1

        updated_record[name_of_student] = presence
        current_row += 1
```



### 6.3. Python Tuples:

Tuples are static, array-like data structures which allow duplicate values. Tuples have been used in SQL Queries because the SQL query functions required that tuples be used.

### 6.4. Python Sets:

Sets are static data structure which neither allow duplicate values nor have an order to its elements. This means none of the elements of the sets can be retrieved using the index value of their elements like with lists, tuples or dictionaries (using keys in this case). Sets have been used to remove duplicate data in lists. By converting a list to a set and back to a list, any duplicate value in the list can be removed.

```
@staticmethod
def process_names(names):
    names = names.strip().split(',')
    for index, name in enumerate(names):
        names[index] = name.strip()
        if name == '':
            names.remove(name)
    names = list(set(names)) # remove repeating names
```

## 7. Object Oriented Programming Features Used

### 7.1. Inheritance

Inheritance in OOP is when a class created based on another existing class. The class which the new class is based on is called the parent while the new class itself is called the child. In this application, especially since all Kivy widgets are classes, a subclass has been created to customize these widgets.

```
class AttendanceApp(App):
    def build(self):
        return kv

class AppScreen(Screen):
    pass

class SecondaryScreens(AppScreen):
    pass

class AttendanceWindow(SecondaryScreens):
    pass

class TakeAttendanceWindow(SecondaryScreens):
    pass
```



## 7.2. Polymorphism

Polymorphism is when the same method is used for different classes/objects. This was used, similar to inheritance, mostly when working with widgets. The same method such as `pos_hint` was used for different widgets.

## 7.3. Classes with Static Methods

In this application, classes have also been used for the purpose of organizing different functions into groups for better code readability maintenance. To do this, static methods (methods that do not require an object instance to be called) have been used as these methods do not require any attribute of the class itself to work.

## 8. Editing the Spreadsheet

The ‘`openpyxl`’ library has been used to add, update and delete data from the spreadsheet. The spreadsheets created have the ‘`.xlsx`’ format.

### 8.1. Inserting New Data Into New Spreadsheet

To insert data into a new spreadsheet, simply the values of particular cells were changed. The name of the spreadsheet file was the name of the record (as specified by the user) followed by ‘`_ta_app`’.

```
@staticmethod
def create_excel_file(names, name_of_record):
    names = FileManager.process_names(names)

    wb = openpyxl.Workbook()
    sheet = wb.active

    # add serial number
    (sheet.cell(row=1, column=1)).value = 'SN'
    for i in range(1, len(names) + 1):
        (sheet.cell(row=i + 1, column=1)).value = i

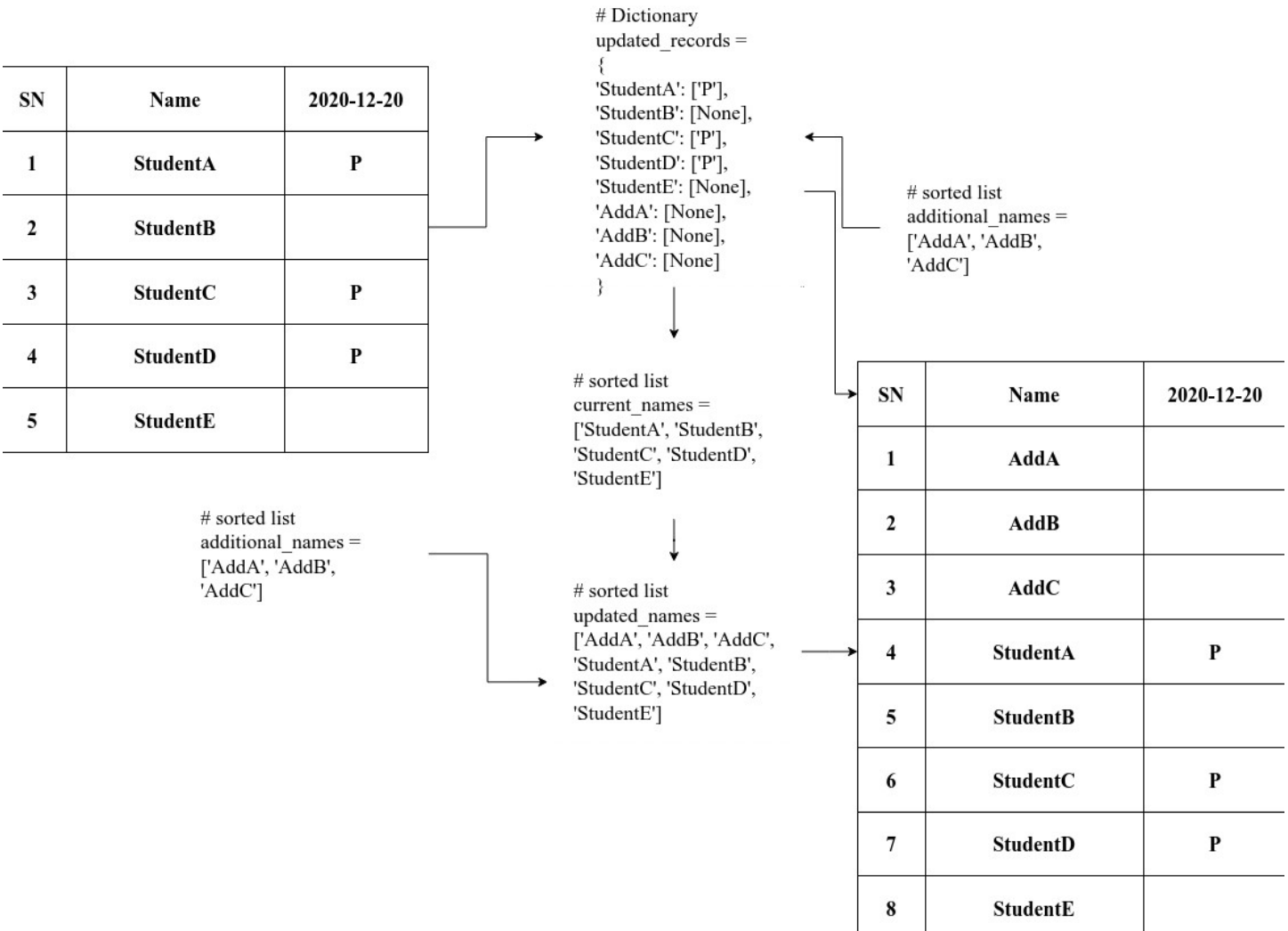
    # add names
    (sheet.cell(row=1, column=2)).value = 'Name'
    for i in names:
        (sheet.cell(row=(names.index(i) + 2), column=2)).value = i

    # save file
    excel_file_path = os.path.expanduser(f'users/{User.current_user}/{name_of_record}_ta_app.xlsx')
    wb.save(excel_file_path)
    FileManager.start_file(excel_file_path)
```

### 8.2. Inserting New Names Into Pre-existing Spreadsheet

In order for the user to be able to insert additional names of students into an already existing record (spreadsheet), an algorithm was designed. Since, this application sorts the names of students in the

spreadsheet, simply adding the additional names was not enough. Below is a diagram which gives a brief overview of the algorithm created. ‘P’ means that the student is present on that particular day. The use of the dictionary here is important as it help store entire rows of the record (the presence of students) making it very easy to swap entire rows when needed.



```

@staticmethod
def add_students_excel(names_of_students, excel_file):
    additional_names = FileManager.process_names(names_of_students)

    wb = openpyxl.load_workbook(excel_file)
    sheet = wb.active

    handler = AttendanceHandler(sheet)
    handler.get_names_excel()

    current_names = handler.excel_names

    updated_record = {}

    max_column = sheet.max_column
    max_row = sheet.max_row

    for name in additional_names:
        updated_record[name] = [None]*(max_column - 2)

    current_row = 2
    while current_row <= max_row:
        name_of_student = (sheet.cell(row=current_row, column=2)).value
        current_column = 3
        presence = []
        while current_column <= max_column:
            presence.append((sheet.cell(row=current_row, column=current_column)).value)
            current_column += 1

        updated_record[name_of_student] = presence
        current_row += 1

    updated_names = [None]*(len(additional_names) + len(current_names))

```

```

# merging the two sorted lists

i = 0
j = 0
k = 0
while i < len(additional_names) and j < len(current_names):
    if additional_names[i] < current_names[j]:
        updated_names[k] = additional_names[i]
        k += 1
        i += 1
    else:
        updated_names[k] = current_names[j]
        k += 1
        j += 1

while i < len(additional_names):
    updated_names[k] = additional_names[i]
    k += 1
    i += 1

while j < len(current_names):
    updated_names[k] = current_names[j]
    k += 1
    j += 1

```

```

# Updated SN
for sn in range(len(current_names), len(updated_names) + 1):
    (sheet.cell(row=sn + 1, column=1)).value = sn

# Updated Names and Presence
for name in updated_names:
    current_row = updated_names.index(name) + 2
    current_column = 2
    (sheet.cell(row=current_row, column=current_column)).value = name
    for presence in updated_record[name]:
        current_column += 1
        (sheet.cell(row=current_row, column=current_column)).value = presence

wb.save(excel_file)
FileManager.start_file(excel_file)

```

### 8.3. Removing Names from Pre-existing Spreadsheet

```
@staticmethod
def remove_students_excel(names_of_students, excel_file):
    names_to_remove = FileManager.process_names(names_of_students)

    wb = openpyxl.load_workbook(excel_file)
    sheet = wb.active

    handler = AttendanceHandler(sheet)
    handler.get_names_excel()

    current_names = handler.excel_names

    updated_record = {}

    max_column = sheet.max_column
    max_row = sheet.max_row

    current_row = 2
    while current_row <= max_row:
        name_of_student = (sheet.cell(row=current_row, column=2)).value
        current_column = 3
        presence = []
        while current_column <= max_column:
            presence.append((sheet.cell(row=current_row, column=current_column)).value)
            current_column += 1

        updated_record[name_of_student] = presence
        current_row += 1
```

```

updated_names = []
for name in current_names:
    if name not in names_to_remove:
        updated_names.append(name)

# Clearing Sheet
for i in range(2, max_row + 1):
    for j in range(1, max_column + 1):
        (sheet.cell(row=i, column=j)).value = None

# Updated SN
for sn in range(1, len(updated_names) + 1):
    (sheet.cell(row=sn + 1, column=1)).value = sn

# Updated Names and Presence
for name in updated_names:
    current_row = updated_names.index(name) + 2
    current_column = 2
    (sheet.cell(row=current_row, column=current_column)).value = name
    for presence in updated_record[name]:
        current_column += 1
        (sheet.cell(row=current_row, column=current_column)).value = presence

wb.save(excel_file)
FileManager.start_file(excel_file)

```

## 8.4. Adding the Presence Status of Students into the Spreadsheet

```

@staticmethod
def update_excel_file(chat_file, excel_file):
    wb = openpyxl.load_workbook(excel_file)
    sheet = wb.active

    take_attendance = AttendanceHandler(sheet)
    present_students = take_attendance.get_present_students(chat_file)

    # making new column and adding current date
    new_column = sheet.max_column + 1
    current_date = str(datetime.date(datetime.now()))
    (sheet.cell(row=1, column=new_column)).value = current_date

    present_students[:] = [student.lower() for student in present_students]

    # adding presence
    for i in range(2, sheet.max_row + 1): # plus one is done as it needs to go up to the last value of sheet.max_row
        if sheet.cell(row=i, column=2).value.lower() in present_students:
            (sheet.cell(row=i, column=new_column)).value = "P"

    wb.save(excel_file)
    FileManager.start_file(excel_file)

```

## 9. Storing and Saving Records Locally

All the spreadsheets (records) are stored locally on the device of the user. In order to ensure that these records are not accessible by an unauthorized user, these records (stored in individual user folders) are password protected and stored in .7z format. The file is unzipped when the user logs in from the application and zipped back when the user logs out. This was done using the 'py7zr' library.

```
@staticmethod
def set_current_user(username, password):
    User.current_user = username
    User.current_user_password = password

    with py7zr.SevenZipFile(f'users/{User.current_user}.7z', 'r', password=f'{User.current_user_password}') as user_dir:
        user_dir.extract()

    os.remove(f'users/{User.current_user}.7z')

@staticmethod
def user_logout():
    with py7zr.SevenZipFile(f'users/{User.current_user}.7z', 'w', password=f'{User.current_user_password}') as user_dir:
        user_dir.writeall(f'users/{User.current_user}')
    shutil.rmtree(f'users/{User.current_user}')
    User.current_user = None
```

## 10. Recent Searches Feature Implementation

The recent searches feature allows the user to view the recently searched terms and has been implemented using a pickle file.

```
@staticmethod
def save_recent_searches(search):
    if not os.path.exists(f'users/{User.current_user}/rsp.pickle'):
        search_data = open(f'users/{User.current_user}/rsp.pickle', 'wb')
        pickle.dump([search], search_data)
        search_data.close()
    else:
        new_searches = [search]
        with open(f'users/{User.current_user}/rsp.pickle', 'rb') as recent_searches:
            for elements in pickle.load(recent_searches):
                new_searches.append(elements)

        with open(f'users/{User.current_user}/rsp.pickle', 'wb') as recent_searches:
            pickle.dump(new_searches, recent_searches)
```



```

@staticmethod
def get_recent_searches():
    if not os.path.exists(f'users/{User.current_user}/rsp.pickle'):
        empty = ['', '', '']
        return tuple(empty)

    searches = []

    with open(f'users/{User.current_user}/rsp.pickle', 'rb') as recent_searches:
        for elements in pickle.load(recent_searches):
            searches.append(elements)

    searches[:] = [search for search in searches if search != '']

    if len(searches) == 1:
        one_search = []
        for elements in searches:
            one_search.append(elements)

        one_search.append('')
        one_search.append('')
        return tuple(one_search)

    if len(searches) == 2:
        two_searches = []
        for elements in searches:
            two_searches.append(elements)
        two_searches.append('')
        return tuple(two_searches)

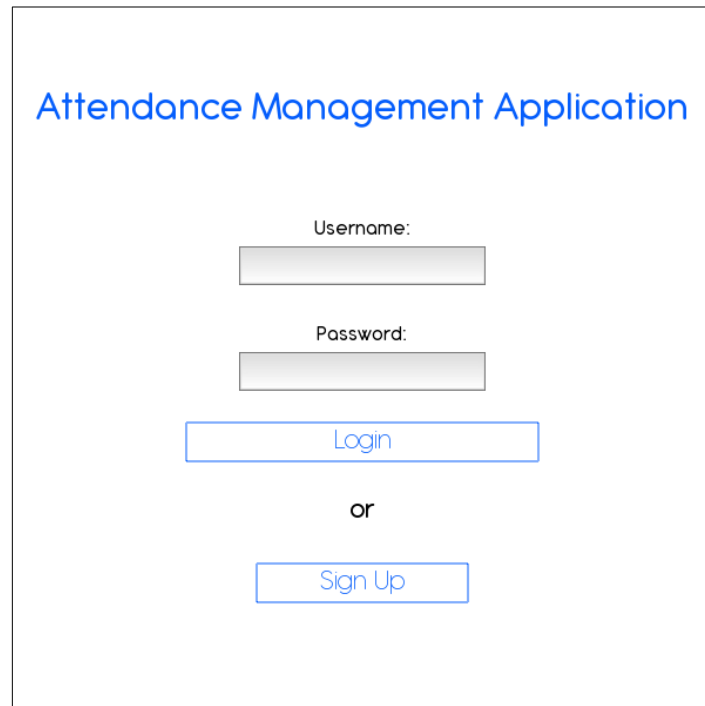
    if len(searches) > 3:
        return tuple(searches[:3])

    return tuple(searches)

```

## 11. GUI Elements

Below are the main UI elements of the application<sup>1</sup>



The image shows a login and sign-up form for an "Attendance Management Application". The form is centered on a light gray background. It features a title "Attendance Management Application" in blue text. Below the title are two input fields: "Username:" and "Password:", each with a corresponding text box. Below these fields is a "Login" button. Below the "Login" button is the word "or". Below "or" is a "Sign Up" button. All buttons and the title text are in blue.

Attendance Management Application

Username:

Password:

Login

or

Sign Up

---

<sup>1</sup> See Appendix B for full code

## Attendance Management Application

New Record

Take Attendance

Update Records

View Records

Help

Logout

## New Record

Name of Students Separated by Comma

Ram Krishna, Hari Bahadur, Sita  
Maya

Name of the Record:

AttendanceRecords

Save

Main Menu

## Attendance Management Application

Username:

Password:

Confirm Password:

Sign Up

Login

## Take Attendance

Upload the chat file.  
Generally, it is located in  
'Zoom' folder in 'Documents'

Browse

Name of the Record You Want to Update:

AttendanceRecords

Take Attendance

Note: You can view the record by going to the  
Main Menu and clicking View Records.

Main Menu

## Update Records

Name of Students Separated by Comma

Ram Krishna, Hari Bahadur, Sita  
Maya

Name of the Record to Update:

AttendanceRecords

Add Students

Remove Students

Main Menu

## View Records

Name of the Record you want to search for:

AttendanceRecords

Search

Recent Searches:

yes

search

test

All Records

Main Menu

## 12. Error Handling and Data Validation

Error handling is important as if the software happens to encounter any errors, the application will not close immediately ruining the experience of the end-user. Hence, the 'try' and 'except' statements of Python have been used to handle errors in the program.

```
@staticmethod
def update(name_of_record, chat_file):
    if '.txt' not in chat_file:
        return 'Upload a chat file and try again!'

    validation_result = Attendance.validate_input(name_of_record)
    if validation_result == 'Successful!':
        try:
            excel_file = FileManager.find_file(f"{name_of_record}", path=f'users/{User.current_user}', specific=True)[0]
            FileManager.update_excel_file(chat_file, excel_file)
            return 'Successful!'
        except IndexError:
            return 'No matching files found!'

    return validation_result
```

For data validation, a function was created for ease

```
@staticmethod
def validate_input(*inputs_to_validate):
    for inputs in inputs_to_validate:
        if type(inputs) == list:
            if any(values for values in inputs if not values.isalpha()):
                return 'Names can only contain letters!'

        else:
            if inputs.isspace():
                return 'Input something and try again!'

            if not inputs.isalnum():
                return 'Field can only contain letters and numbers!'

    return 'Successful!'
```

Here, the \* in front of name of the parameter denotes that any number of arguments can be passed to the function.