Certificate

This is the certificate that

of class _____

of Saraswati Bal Mandir Senior Scondary School

has successfully cubmitted their

Computer Science Project

under the guidance of

Mr. Rahul Bhatia (subject teacher)

during the year 2022-23

as per the examination conducted by AISSCE

Signature of
Internal Examiner

Signature of
External Examiner

For now, this code holds the topic of **Office Management**.
But it is written in such a manner that basically anything similar to this can be build on top of this of same code. It may act like a skeleton for them.
While making this file, our aim was to be as much descriptive as possible, as you will read, you can read our ideology behind every user defined function. And other required info. is mentioned
This code is maintainable, which means, it is written in a manner that it can extended further by anyone who wants to.
It might not be industry standard code but this is written in a proper manner.
The code is divided into multiple files to improve debugging and make each file less dependent and easily editable. First 2 pages contains basic main menu and there isn’t much to explain. So, the actual under the hood code begins from page 3...
None of this code is copy-pasted, the entire code is handwritten.

With all of our hard work and will,

Office Management System in Python

main.py

```python
import datetime
from modules.logs import writeLog, writeUser
from modules.user import User
from modules.admin import matchData
from modules.staticFunctions import printDatabase, printRows, printTables

Errors = User.Errors

class Console:
  def __init__(self) -> None:
    ur = None
    name = ""
    while True:
      mainMenuChoice = int(input('''
              *** Main Menu ***
      1. Login as Guest
      2. Login as Employee
      3. Login as Admin
      4. Exit
      '''))

      if (mainMenuChoice == 1):
        while True:
          try:
            user = str(input("Enter Username : "))
            self.name = user
            passwd = str(input("Enter Password : "))
            self.ur = User(username=user, password=passwd)
            if (self.ur.connection):
              while True:
                try:
                    printDatabase(self.ur.getDatabases())

                    db = str(input("\nSelect a database : "))
                    self.ur.setDatabase(db)
                    writeUser(self.name ,datetime.datetime.now().strftime("%H:%M:%S"), "logged in")
                    print("Successfully Logged in as Guest")
                    break

                except Errors.InvalidDataBaseError:
                  print("Invalid Database")
            break
          except Errors.InvalidUserError:
            print("Invalid Username or Password")

        while True:
          menuChoice = int(input('''
                *** Guest Menu ***
          1 - Fetch All Data
          2 - Search User by ID
          4 - Choose new database
          5 - Back
          '''))

          if menuChoice == 1:
            while True:
              try:
                tables = self.ur.query("SHOW TABLES")
                if tables:
                  printTables(tables)
                  pass
                else: print("No table found")
                table = str(input("\nChoose a table : ")) if tables else "exit"
                if table.lower() == "exit": break
                printRows(self.ur.query("SELECT * FROM " + table))
              except Errors.InvalidSQLQueryError:
                  print("Please choose a valid table")
            continue

          if menuChoice == 2:
            choice = "Y"
            while choice.upper() == "Y":
              id = str(input("Enter ID to be seached : "))
              table = str(input("Enter the name of the Table : "))
              print(self.ur.searchUserByID(id, table))
              choice = str(input("Do you want to search another record? (Y/N) : "))[0]
            pass

```

main.py

```
            if menuChoice == 4:
              while True:
                try:
                  printDatabase(self.ur.getDatabases())
                  db = str(input(str("\nSelect a database : ")))
                  self.ur.setDatabase(db)
                  print("Successfully selected new Database")
                  break
                except Errors.InvalidDataBaseError:
                  print("Invalid Database")

            else: break

        if mainMenuChoice == 2:
          while True:
            try:
              user = str(input("Enter Username : "))
              self.name = user
              passwd = str(input("Enter Password : "))
              self.ur = User(username=user, password=passwd)
              while True and self.ur.connection:
                try:
                  printDatabase(self.ur.getDatabases())
                  db = str(input("\nSelect a database : "))
                  self.ur.setDatabase(db)
                  writeUser(self.name ,datetime.datetime.now().strftime("%H:%M:%S"), "logged in")
                  print("Successfully Logged in as Guest")
                  break

                except Errors.InvalidDataBaseError:
                  print("Invalid Database")
              break
            except Errors.InvalidUserError:
              print("Invalid Username or Password")

        while True:
          menuChoice = int(input('''
                ** Employee Menu **
          1 - Fetch  all Data
          2 - Search User by ID
          4 - Insert Data
          5 - Update a user
          6 - Choose new database
          7 - Back
          '''))

          if menuChoice == 1:
            while True:
              try:
                table = str(input("Enter the name of the table : "))
                if (table.lower() == "exit"): break
                print(self.ur.fetchAllData(table))
              except Errors.InvalidSQLQueryError or TypeError:
                  print("Invalid Syntax, please execute a valid mySQL query")

          if menuChoice == 2:
            while True:
              pass

          if menuChoice == 6:
            while True:
              try:
                printDatabase(self.ur.getDatabases())
                db = str(input("\nSelect a database : "))
                self.ur.setDatabase(db)
                print("Successfully selected new Database")
                break
              except Errors.InvalidDataBaseError or TypeError:
                print("Invalid Database")

            else:
              break

        if mainMenuChoice == 3:
          admin = matchData()
          if (admin):
            admin.proceed()

        if mainMenuChoice == 4:
          exit()

class Gui:
  def __init__(self) -> None:
    pass

Console()
```

<The User 👤/>

This file contains user class...

The ***User*** class takes one's MySQL username and password and instantiates a user with following properties:

- host = “string”
- connection = “MySQL Connection Object”
- _database = “string”
- **__username = “string”
- **__password = “string”

**double underscore(__) is used to specify a private variable as per python's naming conventions.

user.py

```python
import datetime
from typing import Any
from modules.logs import writeLog, writeUser
import mysql.connector as sql

class User:
  host = "localhost"
  connection = sql.connect()
  _database = None
  __username = None
  __password = None
  def __init__(self, username:str, password:str):
    self.__username = username
    self.__password = password

    try:
      self.connection = sql.connect(host=self.host, user=username, passwd=pas
sword)
    except sql.ProgrammingError:
      raise self.Errors.InvalidUserError("Invalid Username or Password")

```

<User Functions/>

---

getDatabases

Special thing about this function is that it doesn't depend on parameters to fetch names of databases from MySQL. And to become independent, it uses the *connection* property of the **User** class.

user.py

```python
# Returns a list of databases
  def getDatabases(self):
    if (self.connection =≠ None):
      cur = self.connection.cursor()
      cur.execute("SHOW DATABASES;")
      data: Any = cur.fetchall()
      return [database for [( database )] in data]
```

setDatabase

All this function does is, it takes the name of the database and sets the *_database* property of **User** and makes a new connection. Using *_database* increases re-usability so, the user won't have to input database every single time a connection is established.

user.py

```python
# Inputs a valid database and creates a new connection with that database
  # If database is invalid, throws "InvalidDatabaseError"
  def setDatabase(self, database:str) → None:
    self._database = database
    try:
      self.connection = sql.connect(host=self.host, user=self.__username, passwd=self.__password, database=self._database)
    except sql.ProgrammingError:
      raise self.Errors.InvalidDataBaseError("Invalid Database")
```

searchUserByID

This function inputs the id and the name of the table and it uses a very simple MySQL query to search search for the very specific user.

user.py

```python
# Inputs ID to be searched and the name of the table to be searched in
def searchUserByID(self, id:str, table:str):
  cur = self.connection.cursor()
  cur.execute("SELECT * FROM {} WHERE no = {}".format(table, id))
  return cur.fetchall()
```

fetchAllData

Working in a very simple way, this function inputs the name of the table to data fetch data from it and returns data in the form of an array.

user.py

```python
def fetchAllData(self, table:str):
  cur = self.connection.cursor()
  cur.execute("SELECT * FROM " + table)
  return cur.fetchall() if cur.fetchall() else []
```

updateUser

This function inputs the name of the table, id of user to be changed and data which is a python dictionary that contains new name, zone and grade.

user.py

```python
def updateUser(self, table:str, id:str, data:dict):
  cur = self.connection.cursor()
  cur.execute("UPDATE {} SET name = '{}', zone = '{}' grade = '{}' WHERE eno = {}".format(table, data["name"], data["zone"], data["grade"], id))
  self.connection.commit()
  pass
```

query

Working in a very simple way, this function inputs the name of the table to data fetch data from it and returns data in the form of a Python list.

user.py

```python
# Takes a mySQL query in the string format as Input
  # Returns a valid output from my mysql (mostly lists)
  def query(self, query:str):
    try:
      if (self.connection):
        cur = self.connection.cursor()
        cur.execute(query)
        writeUser(str(self.__username), datetime.datetime.now().strftime("%H:%M:%S"), "Query : " + query)
        return cur.fetchall() # if cur.fetchall() else []
    except sql.ProgrammingError:
        raise self.Errors.InvalidDataBaseError("No database selected") if (self._database == None) else self.Errors.InvalidSQLQueryError("Invalid Query")

```

user.py

```python
# Errors
class Errors:
  class InvalidUserError(Exception):
    def __init__(self, *args: object) -> None:
      super().__init__(*args)
      # print(Exception)
      writeLog("Invalid User Error faced\n")
  class InvalidDataBaseError(Exception):
    def __init__(self, *args: object) -> None:
      super().__init__(*args)
      # print(Exception)
      writeLog("Invalid Database Error faced\n")
  class UnknownError(Exception):
    def __init__(self, *args: object) -> None:
      super().__init__(*args)
      # print(Exception)
      writeLog("An Unknown Error occured\n")
  class InvalidSQLQueryError(Exception):
    def __init__(self, *args: object) -> None:
      super().__init__(*args)
      # print(Exception)
      writeLog("An Invalid SQL query executed")
```

It's important to handle error throughout the runtime of the program. But sometimes inbuilt errors don't provide a very good Developer Experience and aren't too explicit either.
To improve the Developer Experience of our team, we created custom errors that are as shown...

<Static Funcs/>

Most of the functions you just saw returns data in the for of *list of tuples of string (ex - [(“Actual Data”)])*. And it can be painful to print data in that form. So, to beautify out console and improve user experience, we print data in a much nicer form by use for loops, and to maintain developer experience of our team, we made some special functions solely for this very task.

```python
def printDatabase(databases):
  for database in databases:
    print(database, end = " | ")
  pass

def printTables(tables):
  for [row] in tables:
    print(row, end=" | ")
  input("Enter to continue")
  pass

def printRows(rows):
  for row in rows:
    print(row)
  input("Enter to continue")
  pass
```

Above functions uses for loops and variable de-structuring to perform the task. Our ideology while making these functions was to beautify the terminal and use these functions in the place of primitive *print* function.
Input functions are used so that user can get time to see the previous output instead of instantly pushing next output.

before

```
['blogs', 'garv', 'information_schema', 'kunal', 'mysql', 'performance_schema', 'sakila', 'sys', 'web', 'world']

Select a database :
```

after

```
blogs | garv | information_schema | kunal | mysql | performance_schema | sakila | sys | web | world |
Select a database :
```

<Logging/>

Logging is important to keep track of things that might be important for safety or further development, to do so, we created a few functions to write data in certain files. Some for temporary logs and some of permanent information. All this is done using Python file handling.

```python
import datetime
import csv
import os

logs = open("logs.txt", "w")
logs.write("session started at - " + datetime.datetime
           .now()
           .strftime("%H:%M:%S"))
logs = open("logs.txt", "a")

A_logs = open("./.admin/" + datetime.datetime
              .now()
              .strftime("%d-%m-%y") + ".csv", "a")
adminLogs = csv.writer(A_logs)
# adminLogs.writerow(["NAME", "TIME", "ACTIVITY"])

def writeLog(output:str):
  try:
    logs.write(output)
  except FileNotFoundError:
    print("file not found")

def writeUser(name:str, time:str, activity:str):
  try:
    adminLogs.writerow([name, time, activity])
  except FileNotFoundError:
    print("file not found")

def readLog(name:str) -> list[list[str]]:
  try:
    print(os.getcwd() + "/.admin/" + name + ".csv", "r")
    file = open(os.getcwd() + "/.admin/" + name + ".csv", "r")
    rows = csv.reader(file)
    out = [row for row in rows]
    return out
  except FileNotFoundError:
    print("file not found")
    return []

```

<Admin 👑/>

An Admin plays an important role in the management of a system because one must keep an eye on the what's happening under her/him. Since Admin holds so much power, it is important to keep the position of Admin safe, to do so, we used python's naming convention and named it *__Admin*, ie, it is a private class. Instead of making similar methods for *__Admin* class, a *User* is just instantiated to use all those goods methods we created earlier.

```python
import os
import csv
from modules.logs import writeLog, readLog
from modules.user import writeUser
from modules.user import User
from modules.staticFunctions import printDatabase, printRows

Errors = User.Errors

class __Admin:
  _user = None

  def __init__(self, id:int, passwd:str, user:User) -> None:
    self._user = user
    writeLog("Someone logged in as Admin")
```

To make it even more safer, no other file even touches __*Admin* class, and to make it possible, we make a custom function that do all the validation stuff for initiating __*Admin* and *User* class in a proper manner.

```python
def matchData():

  try:
    adminID = int(input("Enter Admin ID : "))
    adminPasswd = str(input("Enter Password : "))
    adminData = open("./.admin/adminUsers.csv", "r")
    reader = csv.reader(adminData)


    for row in reader:
      if (int(adminID) == int(row[0]) and str(adminPasswd) == str(row[1])):
        while True:
            try:
              user = str(input("Enter Username : "))
              passwd = str(input("Enter Password : "))
              ur = User(username=user, password=passwd)
              break
            except Errors.InvalidUserError:
              print("Invalid Username or Password")
        return __Admin(adminID, adminPasswd, ur)

  except ValueError:
    print("Admin ID must be a password")

  return False
```

For the further menu driven work, the proceed function works out...

Let’s do a break down of further function

```python
def proceed(self) -> None:
  while True:
    menuChoice = int(input('''
          *** Admin Menu ***
    1. Query data from SQL server
    2. Get Login history
    3. User Black List
    4. Switch host
    5. Exit Admin Dashboard
    '''))
```

```
if menuChoice == 1 and self._user:
  try:
    db = None
    while True:
      printDatabase(self._user.getDatabases())
      db = str(input("\nPlease choose a database : "))
      self._user.setDatabase(db)
      break
    while True:
      try:
        query = str(input("Enter a query : "))
        if (query.lower() == "exit"): break
        printRows(self._user.query(query))

      except Errors.InvalidSQLQueryError:
        print("Invalid Query")

  except Errors.InvalidSQLQueryError:
        print("Invalid Database")
```

This choice calls the query method from User class and uses printDatabase function to print it cleanly.

```
if menuChoice == 2:
  dirList = os.listdir("./.admin")
  fileName = str(input(dirList.__str__() + "\nSelect a file : "))

  while True:
    try:
      open("./.admin/" + fileName + ".csv", "r")
      file = readLog(fileName)

      for row in file:
        if row:
          [user, time, status] = row
          print(user + " | " + time + " | " + status)
      input("Enter to continue...")
      break

    except FileNotFoundError:
      print("There is no such file")
```

This choice allows you to get a brief history of users in past days. It reads data from a CSV files that are stored in a very secret place.

And here comes the best example of using classes and it’s properties.

```
1  if menuChoice == 4 and self._user:
2    while True:
3      host = str(input("Enter new hostname : "))
4      self._user.switchHost("")
5      if self._user.connection:
6        print("Connection established Successfully")
7        pass
8      else: print("Invalid Host")
```

Choice 4 allows you to change host of your connection. It is one of the least used but most powerful property of a MySQL connection.

```
1  def switchHost(self, hostname):
2    self.host = hostname
3    self.connection = sql.connect(
4      host=self.host,
5      user=self.__username,
6      passwd=self.__password
7      )
```

Now this function is a part of ***User*** class and is a perfect example of using properties, this time the program already had all those required values and saves time.