Student Name: **Weight:** **30%**

Student ID: **Marks:** **/100**

Assignment: Classes

Type: Group Assignment – Covers Topics from Units 1 - 5

- **Students should ONLY use programming constructs covered in the course units**.
    - **Submission will not be accepted if programming concepts that are not covered in course units are used.**
- **Late submissions will not be accepted.**
- **Individuals will not receive a grade on the group assignments if they do not complete peer assessment and demonstration video.**

Scenario

Alberta Rural Hospitals (ARH) is a new healthcare provider in Alberta. To complement the existing large-scale hospitals located in urban settings, ARH is building a network of smaller scale mini-hospitals which target underserved rural populations. ARH has hired your company to create a management system which is customized to meet their unique operational needs. Your group will work on Phase 1 of the project.

Equipment and Materials

For this assignment, you will need:

- Python IDE
- GitHub repository

Instructions

This assignment consists of 3 sections, all completed outside of class time. See the course schedule and Brightspace for exact dates.

Git/GitHub Training LinkedIn Learning (5%)

Learning how to work efficiently with a team in a hosting platform such as GitHub is an essential skill for programmers. A group coding project such as this one provides the perfect opportunity to learn about and then practice this essential skill.

1. Complete one of the approved LinkedIn Learning courses
2. Submit a copy of your certificate of completion to Brightspace.

Note: There should be no out of pocket expenses for the LinkedIn Learning course. As a SAIT student, you have free access to thousands of professional development courses through LinkedIn Learning. Ask your instructor if you run into issues accessing the courses.

3. Have one member of your group set up a GitHub repository for this project. Make sure that this GitHub repository is **private**. Add group members and your instructor as collaborators.
4. Create a separate branch in GitHub for each group member, containing the part they work on. The branch name should include the task name and the student name.
5. GitHub must be effectively used for group collaboration.
6. Ensure all group members push their parts to their respective GitHub branches.
7. The final project code must be pushed to master/main on GitHub.

Project Submission – Group (90%)

1. Check your solution against the detailed marking criteria at the end of this document.
2. Submit the following files to Brightspace:
    - The code of the program that you implemented (.py file).
    - A copy of the test output (.txt file)
    - All data files updated by program after the test run has been completed.
    - A link to your GitHub project repository

Individual Submission (5%)

1. **Peer Assessment (1.0%)**

   Each student must also complete a peer assessment of their group members.
    - In the peer assessment, each group member will submit a quiz questionnaire for the other group members (not themselves).
    - Quiz questionnaire is in Brightspace under Assessments-Tests.

2. **Demonstration Video (4.0%)**

   Each student will submit a **video** (no more than 5 minutes long) to Brightspace. In the video, the student will:
    - introduce themselves and indicate which team they are a part of (i.e. Team number and group members) – camera should be on. Please use laptop camera/screen share to create video (not phone).
    - share the **group's final solution** (Python code) and explain a **part of the code** that they developed. Debugger may be used to step through code (optional). This is **not** to be **a demo** of running the application.

Management System Details

Alberta Rural Hospitals (ARH) requires that their management system application meets the following criteria.

- Supports data entry as well as report generation
- Uses the following classes throughout the application:
  - #1 - Doctor
  - #2 - Patient
- Consists of **two** separate modules:
  1) module containing the classes
  2) module containing the application that uses the classes
- Uses the methods/functions listed below
- Each object has descriptive properties/ attributes that represent the work and actions of the class as outlined below.

Class #1: Doctor

Properties/Attributes (hidden)

doctor id, name, specialty, schedule, qualification, room number

Methods

Method Name	Description
Constructor	• __init__() should initialize the Doctor object properties. • The constructor should allow creating a doctor object without passing values to the constructor o Hint: Use keyword arguments in the constructor
Getters	• Implement one getter function for each Doctor property. The getter function should return the value of the property o E.g. get_doctor_id (self)
Setters	• Implement one setter function for each Doctor property. The setter function should set the property to a new value o E.g. set_doctor_id (self, new_id)
format_file_layout()	• Returns a string representation of a Doctor object • This representation should include all doctor properties separated by an underscore (-) • Same format as “doctors.txt” file

__str__()	• Returns a string representation of a Doctor object • This representation should display the doctor properties as shown in the Sample Run, “Display Doctor’s List”

Sample data: doctors.txt (data file provided)

Class #2: Patient

Properties/Attributes (hidden)

patient id, name, diagnosis, gender, age

Methods

Method Name	Description
Constructor	• __init__() should initialize the Patient object properties. • The constructor should allow creating a patient object without passing values to the constructor o Hint: Use keyword arguments in the constructor
Getters	• Implement one getter function for each Patient property. The getter function should return the value of the property o E.g. get_patient_id (self)
Setters	• Implement one setter function for each Patient property. The setter function should set the property to a new value o E.g. set_patient_id (self, new_id)
format_file_layout()	• Returns a string representation of a Patient object • This representation should include all patient properties separated by an underscore (-) • Same format as “patients.txt” file
__str__()	• Returns a string representation of a Patient object • This representation should display the patient properties as shown in the Sample Run, “Display Patient’s List”

Sample data: patients.txt (data file provided)

Management Module

Create a **separate module** for your management application code which will be comprised of various functions, including a **main()** entry function that controls the overall processing. The functions listed below are required but you may use additional functions too.

Doctor-related Functions

Function Name	Description
manage_dr()	• Creates an empty list for Doctor objects • Reads doctors file • Manages all the doctor menu options • Writes doctors file
read_doctors_file()	• Reads data from file “doctors.txt” • Creates Doctor objects for each doctor record • Appends object to doctor list
find_dr_by_id()	• Receives the doctor ID to locate • Searches the list of Doctor objects for a specified doctor with specified ID • If found, prints and returns Doctor object • Otherwise returns -1.
match_dr_by_name()	• Asks the user to enter the name/partial name to match • Searches the list of Doctor objects for a name that contains the name requested • Displays each Doctor objects that match criteria • If no match, prints not found message.
edit_dr_info()	• Asks the user for the doctor ID to edit • Checks to see if that doctor ID exists in list of Doctor objects • If it does not exist, prints an error message • Otherwise, prompts for new values and updates the Doctor object with new values • Displays doctor list
display_list_of_drs()	• Displays all the Doctors’ information(attributes) in doctors list in the format specified in the Sample run
write_drs_list_to_file()	• Writes “doctors.txt” file from the list of Doctor objects, maintaining correct formatting

add_dr_to_list()	• Asks the user to enter a doctor id • Checks to see if that doctor ID exists in list of Doctor objects • If it does exist, prints error message • Otherwise, Asks the user to enter the rest of the doctor information (attributes), creates a new Doctor object (with user-entered doctor information) and adds it to the doctors list

Patient-related Functions

Function Name	Description
manage_patient()	• Creates an empty list for Patient objects • Reads patients file • Manages all the patient menu options • Writes patients file
read_patients_file()	• Reads data from file “patients.txt” • Creates Patient objects for each patient record • Appends object to patients list
find_patient_by_id()	• Receives the Patient ID to locate • Searches the list of Patient objects for a patient with specified ID • If found, prints and returns Patient object • Otherwise returns -1.
edit_patient_info()	• Asks the user for the Patient ID to edit • Checks to see if that Patient ID exists in list of Patient objects • If it does not exist, prints an error message • Otherwise, prompts for new values and updates the Patient object with new values • Displays patient list
display_list_of_patients()	• Displays all the Patients’ information(attributes) in patients list in the format specified in the Sample run
write_patients_list_to_file()	• Writes “patients.txt” file from the list of Patient objects, maintaining correct formatting
add_patient_to_list()	• Asks the user to enter a patient ID

	• Checks to see if that patient ID exists in list of Patient objects • If it does exist, prints error message • Otherwise, Asks the user to enter the rest of the patient information (attributes), creates a new Patient object (with user-entered patient information) and adds it to the patients list

Hint: calling many of the management functions will require that you pass the list of doctor or patient objects as an argument.

Menu Functions

Function Name	Description
menu()	• Receives a menu name and menu dictionary • Displays specified menu, • Accepts menu selection from user until valid selection is entered • Returns user's **valid** selection

Sample Run: see file *W2023 Project Sample Run.pdf*.

Marking Criteria

Individual Submissions

		Marks
Git/GitHub training	• Submission of LinkedIn Learning certificate of completion for either Git or GitHub course	**/5**
Peer assessment and demonstration video	• Review the above peer assessment and demonstration video section	**/5**

Group Submission

	Needs Improvement (0–50%)	Good (51–75%)	Excellent (76–100%)	Marks
Working code	• The project doesn't run in all scenarios • Input requests work but don't match the scenario • No conversion of data types • Did not follow best practices for loops/conditions • Syntax of if/else statements has mistakes • Use of classes is poor • Use of functions is poor • Output works but doesn't match the scenario • Unable to read/write to files, or with many errors	• The project runs in all scenarios • Input requests work but don't match the scenario • Some functions or methods are missing • Did not follow best practices for loops/conditions • Correct use of if/else statements • Correct use of classes • Output works but doesn't completely match the scenario • Able to read/write to files but with a minimal errors	• The project runs in all scenarios • Input requests match the scenario exactly • Correct functions and methods developed • Correct use loops/conditions • Correct use of if/else statements • Correct use of classes • Output matches the scenario • Able to read and write to files correctly – open/close files	**/55**
Style	• Indentation – not consistent • Readability – poor variable names and no use of whitespace • Some naming standards for modules, variables, constants, functions/methods, classes not followed	• Indentation – some parts are consistent and some are not • Readability – some variable names are not descriptive, poor use of whitespace • Some naming standards for modules, variables, constants, functions/methods, classes not followed	• Indentation – consistent • Readability – descriptive variable names, good use of whitespace • Uses correct naming standards for modules, variables, constants, functions/methods, classes	**/20**

	• Documentation o No comments are included at the top. o No comments indicating major code sections or what they do o No function/method documentation (docstring)	• Documentation o Comments at the top are missing or incomplete. o Comments indicating major code sections and what they do are incomplete o Incomplete function/method documentation (docstring)	• Documentation o Comments at the top are complete and include name, date, program description including details on inputs, processing and outputs (4–5 sentences minimum). o Comments indicate major code sections and what they do o Mostly complete functions/methods documentation (docstring)	
Testing	• Sample output, produced by executing program missing most of Sample Run Test Cases provided • Output is not formatted according to the specification (sample run) • Results hard-coded in program rather than implemented with logic	• Sample output, produced by executing program missing some of Sample Run Test Cases provided • Output formatted according to the specification (sample run)	• Sample output, produced by executing program, closely matches the Sample Run Test Cases provided • Output formatted according to the specification (sample run)	**/10**
Version control (evaluated in Github)	• No evidence that group members practiced version control best practices	• Some evidence that some group members adhered to version control best practices.	• It is evident that all group members are consistently adhering to version control best practices.	**/5**
			Total	**/90**