ACIT 2515 – Object Oriented Programming - Lab 3
Encapsulation and Abstraction (Friday Set Only)

Instructor	Mike Mulder (mmulder10@bcit.ca) Also available on Slack.
Total Marks	25
Due Dates	Thursday, Jan. 31st at midnight

Goals

- Apply the Object Oriented Programming Principles of Encapsulation and Abstraction.
- Continue to exercise good Python programming practices including naming conventions, documentation and unit testing.

Overview

The sensor_results.py module is a Python script that loads in temperature sensor readings from a .csv (comma separated values) file. It loads the sensor readings into a "list of lists". It then processes the sensor readings and outputs a report of the results to the console.

The script was written by a former co-worker who is no longer at the company you work for, ABC Sensors. ABC Sensors has plans to build a new product that will use the functionality in the script. Because you know this script is going be used heavily in the future and extended to handle multiple types of sensor readings, you want to start using OOP to make it more reusable and maintainable.

Today you are going to create a class to hold a single sensor reading and integrate this new class into the script.

CSV File

The sensor_results.py script processes temperature sensor readings from a .csv file (sensor_results.csv). The .csv file contains sensor readings in the following format (where each line is a separate reading):

<DateTime>,<Sequence #>,<Sensor Model>,<Low Temp>,<Avg Temp>,<High Temp>,<Status>

Where:

- DateTime – The date and time of the reading (to the fraction of a second)
- Sequence # – Identifies the order of the readings (starting at 1) in the file
- Sensor Model – The name of the ABC Sensor temperature sensor model
- Low Temp – The lowest temperature over the reading period
- Avg Temp – The average temperature over the reading period

- High Temp – The highest temperature over the reading period
- Status – OK, LOW_TEMP or HIGH_TEMP. LOW_TEMP means below the sensor temperature limit (-50°C) and HIGH_TEMP means above the sensor temperature limit (100°C).

Example:

```
2018-09-23 19:56:01.003,1,ABC Sensor Temp M301A,20.112,21.345,22.003,OK
2018-09-23 19:57:02.234,2,ABC Sensor Temp M301A,-50.000,-50.000,-50.000,LOW_TEMP
2018-09-23 19:58:01.877,3,ABC Sensor Temp M301A,100.000,100.000,100.000,HIGH_TEMP
```

The sensor_results.py script loads each line from the sensor_results.csv file into a list where the elements in the list match the order of the data in the .csv file:

["2018-09-23 19:56:01.003",1,"ABC Sensor Temp M301A",20.112,21.345,22.003,"OK"]
["2018-09-23 19:57:02.234",2,"ABC Sensor Temp M301A", -50.000,-50.000,-50.0,"LOW_TEMP"]
["2018-09-23 19:57:01.877",3,"ABC Sensor Temp M301A", 100.000,100.000,100.0,"HIGH_TEMP"]

Part A – Class Design (5 marks)

Using **UML Class Notation**, design a class for a sensor reading using the OOP principles of encapsulation and abstraction. Note: Name the class SensorReading and include the constructor as a method in your UML definition.

Your design should NOT be purely getter methods for each of the instance variables. It should provide a public interface that is useful when integrated into the sensor_results.py script.

Hints:

- Store the DateTime as a Python datetime in your sensor reading class rather than a string. This way the internal representation has the actual datetime value but the public interface has flexibility in how it provides the datetime (i.e., as a formatted string).
- In addition to accessor methods, include methods that do useful thing currently done by the sensor_results script (i.e., calculate the temperature range, format output strings such as error messages, indicate whether the reading is an error reading)

Part B – Class Implementation and Test (15 marks)

Implement a sensor reading class in Python that matches your design (**10 marks**). Your class must include:

- Constants
- Private Instance Variables
- Constructor/Initializer
- Public Methods
- Private Methods (instance or static, if appropriate)

- Parameter Validation (for any public methods, including the constructor, that take in parameters). An invalid parameter should raise a ValueError (see Lab 2 for an example).

Implement a unit test for your Python class using the unittest framework (**5 marks**). Your unit test class must include:

- At least one "success" test and one "alternate"/"error" test (only if applicable) per public method.
- setUp and tearDown methods to create a test fixture (if applicable for your tests) and print out a "logPoint" message before/after each test (see lecture notes) to improve traceability. See the Week 3 lecture slides for details on setUp, tearDown and logPoint.

Your Lab 3 project in PyCharm (or other IDE) should contain the following files:

- sensor_results.py (provided)
- sensor_results.csv (provided)
- sensor_reading.py (contains your SensorReading class implementation)
- test_sensor_reading.py (contains the unit tests for your SensorReading class)

Part C – Integration (5 marks)

In the sensor_results.py script, refactor the "list of lists" for the sensor readings into a "list of sensor reading objects". Make sure everything that used the previous sensor reading data from the list is now using your sensor results class. Note: You will need to import your SensorReading class into sensor_results.py in order to be able to use it. Make sure you follow the best practice described in today's lecture.

Make sure all the outputs from the sensor_results.py script from before and after your refactoring are exactly the same.

Grading Summary

Part A – Class Design (in UML Class Notation) • Class Name (1 mark) • Attributes (2 marks) • Methods (2 marks)	5 marks
Part B – Class Implementation and Test • Sensor Result Class (10 marks) • Unit Test Class (5 marks)	15 marks
Part C – Integration • Refactoring to use Sensor Result Class (3 marks) • No Change in Outputs (2 marks)	5 marks
Marks will be subtracted poor programming practices, including: • Violations of naming conventions • Missing or invalid DocString • Missing parameter validation • Failing unit tests • Unnecessary print statements left in code Note: Not applicable to the sensor_results.py script.	-1 mark each
Total	**25 marks**

Submission

Upload the following to D2L in a **zipfile** called **lab3.zip** (Activities -> Assignments -> Lab 3):

- The file containing your UML Class Diagram for the sensor class (**sensor_reading.pdf**)
- The file containing your sensor reading class (**sensor_reading.py**)
- The file containing your sensor reading unit test (**test_sensor_reading.py**)
- The updated **sensor_results.py** script

Note that the sensor_results.csv file should NOT require updates for this lab.

Documentation Best Practices

Use the following documentation practices below for this lab.

Class Documentation	Add a comment describing what the class represents.
	
	Use DocString as per the following example:
	
	`class Point:`
	`  """Represents a point in 2D geometric coordinates"""`
	
	`  def __init__(self, x=0, x=y):`
	`  ...`

Method Documentation	Add a comment describing what the method does.
	
	`def __init__(self, x=0, x=y):`
	`  """Initialize the position of a new point. The x and y`
	`  Coordinates can be specified. If they are not, the`
	`  point defaults to the origin. """`
	
	`def move(self, x, y):`
	`  """Move the point to a new position in 2D space. """`
	`  self.x = x`
	`  self.y = y`

Docstring Reference: https://www.python.org/dev/peps/pep-0257/

Naming Best Practices

Use the following naming practices for this lab.

Class Name	CapitalizedWords (aka CamelCase)
Instance Variables	lower_case_with_underscores **Note:** Use _lower_case_with_underscores for internal (i.e., private) instance variables.
Methods	lower_case_with_underscores **Note:** Use _lower_case_with_underscores for internal (i.e., private) methods.

Reference Style Guide: https://www.python.org/dev/peps/pep-0008/