Assignment 4 – Solutions and Grading

ADTs and Objects

Date Due: Wednesday, June 16, 11:59pm **Total Marks: 40**

Question 1 (10 points):

Learning Objectives

Purpose: Students will practice the following skills:

- Designing test cases for a given ADT.
- Understanding the nature of errors in floating point calculations!
- Working with an object-oriented ADT.
- Increased work speed. Writing test cases should be fast and thorough.

Degree of Difficulty: Easy.

References: You may wish to review the following:

- Chapter 7: Testing
- Chapter 8: ADTs
- Chapter 9: Objects
- Page **??** on floating point calculations

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

On the course Moodle, you'll find:

- The file `Statistics.py`, which is an ADT covered in class and in the readings. For your convenience, we removed some of the calculations and operations (e.g., `var()` and `sampvar()`, that were not relevant to this exercise, which would have made testing too onerous.
- The file `test_statistics.py`, which is a test-script for the `Statistics` ADT. This test script currently only implements a few basic tests.

In this question you will complete the given test script. Study the test script, observing that each operation gets tested. Because each object is different we cannot check if an object is created correctly; instead, we will check if the initial values are correct. Likewise, methods like `add()` can't be tested directly, because the object's attributes are private; all we can really do is check to see if the add() method has the right effect in combination with `count()` and `mean()`.

Design new test cases for the operations, considering:

- Black-box test cases.

- White-box test cases.
- Boundary test cases, and test case equivalence classes.
- Test coverage, and degrees of testing.
- Unit vs. integration testing.

Running your test script on the given ADT should report no errors, and should display nothing except the message `'*** Test script completed ***'`.

Note: You will have to decide how many tests to include. This is part of the exercise.

What to Hand In

- A Python script named `a4q1_testing.py` containing your test script.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 5 marks: Your test cases for `Statistics.add()` have good coverage.
- 5 marks: Your test cases for `Statistics.mean()` have good coverage.

Solution: A model solution, developed on the basis of the given file, can be found on the Moodle page: `a4q1_testing.py`. The test script was modified to use absolute difference (the function `close_enough()`, rather than exact equality, when checking the mean.

Because the class definition made all the attributes private, we can't check their value without calling a method. As a result, we can claim to be doing unit testing. We're focussed on a single class, but using multiple methods in each test. So by definition, we're doing a simple form of integration testing.

The given file `test_statistics.py` had three sets of tests.

- Initialization. When a new Statistics object is created, ensure that the `count()` and `mean()` method return the correct initial values, according to their documentation.
- `add() + count()`. A couple of simple tests checking whether these two methods work consistently together.
  1. A test case for a single data value. This is a boundary case. It's the smallest example of adding data.
  2. A test case for multiple values. It's an example in the equivalence class of multiple values.
- `add() + mean()` A couple of simple tests checking whether these two methods work consistently together.
  1. A test case for a single data value. This is a boundary case. It's the smallest example of adding data.
  2. A test case for multiple values. It's an example in the equivalence class of multiple values.

I added the following test cases:

- Initialization. None. There is no possibility that more test cases of creating a new object can reveal new errors.
- `add() + count()`. I added a single test case, but only to live up to my "2 examples per class " suggestion:
  1. A test case for multiple values. It's an example in the equivalence class of multiple values.
- `add() + mean()` I added a bunch of examples here:
  1. Equivalence class: Positive floating point values only
  2. Equivalence class: Negative floating point values only
  3. Equivalence class: Mixed floating point values only
  4. Equivalence class: All the same value
  5. Equivalence class: Extreme (very large and very small) floating point values only. This is an example of something most first year students wouldn't think to test! It goes beyond the normal or expected examples. It turns out that my simple test script produces a failed test case here, but that's a test script error not a bug in the class. The error is related to the size of the values being tested. The values in this test are so large that the very small difference used in the call to `close_enough()` cannot be satisfied. A larger value for the acceptable difference will show that the class adds correctly. The very next test demonstrates this.

The question mentions coverage. Because the class is small, and the methods short, it was easy to get 100% coverage:

- Every method was covered.
- Every line of code was covered.

However, we cant test `mean()` without using `add()` because of the design of the class, so the mark here should be given out of 10 for coverage in general.

Notes for markers:

- The number of test cases can be a little smaller than the model solution demonstrates.
- My experience says that further testing of initialization and `count()` is not necessary, but don't deduct marks for having more test cases there.
- The main criterion for test cases is that they seem deliberately chosen, and not random. Give full marks (5) if there are several (3 or more) test cases that don't seem random. Things like this are acceptable:
  - Positive integers; negative integers
  - Positive floating point; negative floating point
- It's not necessary to have the same kinds of examples as the model solution.
- The reasons stated in the test case may or may not be good; that's okay.
- Hopefully the students used the absolute difference rather than exact equality. But this is not worth marks, so don't deduct marks if they use equality.

Question 2 (15 points):

Learning Objectives

Purpose: Students will practice the following skills:

- Reading and understanding an ADT written as a Python class.
- Adding operations to an existing ADT, without breaking what's already there.
- Adding test cases for new operations, to ensure that no bugs are introduced.
- Increased work speed. Writing test cases should be fast and thorough.

Degree of Difficulty: Easy.

References: You may wish to review the following:

- Chapter 7: Testing
- Chapter 8: ADTs
- Chapter 9: Objects

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

On the course Moodle, you'll find:

- The file `Statistics.py`, which is an ADT covered in class and in the readings. For your convenience, we removed some of the calculations and operations (e.g., `var()` and `sampvar()`, that were not relevant to this exercise, which would have made testing too onerous.

In this question you will define two new operations for the ADT:

- `maximum()` Returns the maximum value ever recorded by the Statistics object. If no data was seen, returns `None`.
- `minimum()` Returns the minimum value ever recorded by the Statistics object. If no data was seen, returns `None`.

Hint: To accomplish this task, you may have to modify other operations of the `Statistics` ADT.

You will also improve the test script from Q1 to test the new version of the ADT, and ensures that all operations are correct. Remember: you have to test all operations because you don't want to introduce errors to other operations by accident; the only way to know is to test all the operations, even the ones you didn't change. To make the marker's job easier, label your new test cases and scripts so that they are easy to find.

What to Hand In

- A text file named `a4q2_changes.txt` (other acceptable formats) describes changes you made to the existing ADT operations. Be brief!
- A Python program named `a4q2.py` containing the ADT operations (new and modified), but no other code.

- A Python script named `a4q2_testing.py` containing your test script.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of all documents.

Evaluation

- 2 marks: Your added operation `maximum()` is correct and documented.
- 2 marks: Your added operation `minimum()` is correct and documented.
- 3 marks: Your modifications to other operations are correct.
- 8 marks: Your test script has good coverage for the new operations.

Solution: A model solution for the ADT operations can be found in the file `a4q2.py`, and a corresponding test script is found in `a4q2_testing.py`.

New Operations There are essentially two ways to implement minimum and maximum:

- Keep track of a single max value and a single min value, the same way that the count and average are updated.
- Keep all values in a list and recompute max and min as needed. This is highly inefficient!

The model solution shows that one can add two new attributes to the record: `__max` and `__min`. In `__init__()`, these new attributes are set to `None`. Using `None` in this way helps keep track of whether any data has been added yet. Depending on the design of the new methods, using another value here is acceptable.

```python
    def __init__(self):
        """
        Purpose:
            Initialize a Statistics object instance.
        """
        self.__count = 0       # how many data values seen so far
        self.__avg = 0         # the running average so far
        self.__max = None       # the biggest value seen so far
        self.__min = None       # the smallest value seen so far
```

The max and min values are updated in the `add()` operation. Basically, compare the current max (and min) to the new value being added, and remember the bigger one. One has to take care when no data has been added, because the initial min and max are `None`. In that case, the new value is both the min and the max.

```python
    def add(self, value):
        """
        Purpose:
            Use the given value in the calculation of mean and
            variance.
        Pre-Conditions:
            :param value: the value to be added
        Post-Conditions:
            none
        Return:
            :return none
        """
        # update the running count
        self.__count += 1

        # update the running average
        diff = value - self.__avg  # convenience
        self.__avg += diff / self.__count

        # update the running max
        if self.__max is None or self.__max < value:
            self.__max = value
        else:
            pass

        # update the running min
        if self.__min is None or self.__min > value:
            self.__min = value
        else:
            pass
```

Since all the work is done in `add()`, the operations `minimum()` and `maximum()` simply return the recorded value.

```
    def maximum(self):
        """
        Purpose:
            Return the maximum of all the values seen so far.
        Post-conditions:
            (none)
        Return:
            The mean of the data seen so far.
            Note: if no data has been seen, None is returned.
        """
        return self.__max
```

This combination of operations working together is very efficient.

The alternative implementation, keeping a list of data, means that `add()` is a bit simpler for the programmer, but the operations `minimum()` and `maximum()` are a lot more expensive: using `min()` and `max()` is linear search, which is much more expensive than returning a stored value.

The efficient solution is so much better that using the inefficient method will incur a deduction of one point out of 10. Enough to get the attention of students, but not enough to affect anyone's grade.

Testing The assignment mentions coverage again. All the new operations should be tested in the new script. That's about all we can ask for.

I added three new sets of tests:

- Initialization. I checked that the when no data had been added, the two methods return `None`.
- `maximum()`: I checked the following equivalence classes:
  - Boundary case: One data value added.
  - Multiple data values added, all positive
  - Multiple data values added, all negative
  - Multiple data values added, mixed positive and negative
  - Multiple data values added, mixed positive and negative, largest first
  - Multiple data values added, mixed positive and negative, largest last
  - Multiple data values added, mixed positive and negative, largest midway
- `minimum()`: I checked the analogous equivalence classes for the minimum operation.
  - Boundary case: One data value added.

- Multiple data values added, all positive
- Multiple data values added, all negative
- Multiple data values added, mixed positive and negative
- Multiple data values added, mixed positive and negative, smallest first
- Multiple data values added, mixed positive and negative, smallest last
- Multiple data values added, mixed positive and negative, smallest midway

I do not expect students to be this thorough this early in their career. This solution is intended to show how I am thinking, not expectations on the students' work.
Notes for markers:

- Implementation of the methods:
  - 2 marks: The added operation `maximum()` is correct and documented.
  - 2 marks: The added operation `minimum()` is correct and documented.
  - Both implementations can be considered correct for these points. The deduction for inefficiency is below.
- Other modifications:
  - Specifically, `add()` and `__init__()`.
  - 3 marks: The modifications to other operations are correct. Give a one mark deduction for the inefficient implementation here.
  - Testing:
    - 8 marks: Your test script has good coverage for the new operations.
    - The easiest way to add test cases is to use the test cases from A4Q2, and simply call the new methods on those. That's worth full marks.
    - If the testing of the new methods was incomplete, e.g., they only had one or two examples for each method, then deduct 4 marks.
    - If the testing did not test the new methods, give zero.

Question 3 (15 points):

Learning Objectives

Purpose: Students will practice the following skills:

- Reading and understanding code written by another developer.
- Modifying and adapting the given code to increase functionality.
- Practice concepts related to Python classes and objects.

Degree of Difficulty: Easy.

References: You may wish to review the following:

- Chapter 8: ADTs
- Chapter 9: Objects

Restrictions: This question is homework assigned to students and will be graded. This question shall not be distributed to any person except by the instructors of CMPT 145. Solutions will be made available to students registered in CMPT 145 after the due date. There is no educational or pedagogical reason for tutors or experts outside the CMPT 145 instructional team to provide solutions to this question to a student registered in the course. Students who solicit such solutions are committing an act of Academic Misconduct, according to the University of Saskatchewan Policy on Academic Misconduct.

In this question we will be working with 2 object-oriented classes, and objects instantiated from them, to build a tiny mock-up of an application that a teacher might use to store information about student grades.

Obtain the file `a4q3.py` from Moodle, and read it carefully. It defines two classes:

- `GradeItem`: A grade item is anything a course uses in a grading scheme, like a test or an assignment. It has a score, which is assessed by an instructor, and a maximum value, set by the instructor, and a weight, which defines how much the item counts towards a final grade.
- `StudentRecord`: A record of a student's identity, and includes the student's grade items.

At the end of the file is a script that reads a text-file, and displays some information to the console. Right now, since the program is incomplete, the script gives very low course grades for the students. That's because the assignment grades and final exam grades are not being used in the calculations.

Complete each of the following tasks:

1. Add an attribute called `final_exam`. Its initial value should be `None`.
2. Add an attribute called `assignments`. Its initial value should be an empty list.
3. Change the method `StudentRecord.display()` so that it displays all the grade items, including the assignments and the final exam, which you added.
4. Change the method `StudentRecord.calculate()` so that it includes all the grade items, including the assignments and the final exam, which you added.
5. Add some Python code to the end of the script to calculate a class average.

Additional information

The text-file `students.txt` is also given on Moodle. It has a very specific order for the data. The first two lines of the file are information about the course. The first line indicates what each grade item is out of (the maximum possible score), and the second line indicates the weights of each grade item (according to a grading scheme). The weights should sum to 100. After the first two lines, there are a number of lines, one for each student in the class. Each line shows the student's record during the course: 10 lab marks, 10 assignment marks, followed by the midterm mark and the final exam mark. Note that the marks on a student line are scores out of the maximum possible for that item; they are not percentages!

The function `read_student_record_file(filename)` reads `students.txt` and creates a list of `StudentRecords`. You will have to add a few lines to this function to get the data into the student records. You will not have to modify the part of the code that reads the file.

List of files on Moodle for this question

- `a4q3.py` — partially completed
- `students.txt`

What to Hand In

- The completed file `a4q3.py`.

Be sure to include your name, NSID, student number, course number and laboratory section at the top of the file.

Evaluation

- 1 mark: You correctly added the attribute `final_exam` to `StudentRecord`
- 1 mark: You correctly added the attribute `assignments` to `StudentRecord`
- 4 marks: You correctly modified `StudentRecord.display()` correctly to display the new grade items.
- 4 marks: You correctly modified `StudentRecord.calculate()` correctly to calculate the course grade using the new grade items.
- 5 marks: You added some code to the script that calculates the class average.

Solution:

- Adding GradeItems to the StudentRecord.

```
class StudentRecord(object):
    def __init__(self, first, last, st_number):
        """
        Purpose:  Initialize the Student object.
        Preconditions:
            :param first: The student's first name, as a string
            :param last: The student's last name, as a string
            :param st_number: The student's id number, as a string
        """
        self.first_name = first
        self.last_name = last
        self.id = st_number
        self.midterm = None
        self.final = None
        self.labs = []
        self.assignments = []
```

  The initial values are `None` and an empty list.

- Modifying `StudentRecord.display()`

```
    def display(self):
        """
        Purpose:
            Display the information about the student, including
            all lab and assignment grades, and the calculation of
            the total grade.
        Return:
            None
        """
        print("Student:", self.first_name, self.last_name,
              '(' + self.id + ')')
        print("\tCourse grade:", self.calculate())
        print('\tMidterm:', str(self.midterm))

        print('\tFinal:', str(self.final))
        print('\tLabs:', end=" ")
        for g in self.labs:
            print(str(g), end=' ')
        print()
        print('\tAssignments:', end=" ")
        for g in self.assignments:
            print(str(g), end=' ')
        print()
```

  The display should be using the Python `str()` function. For the list of assignments, I believe the simplest solution is to copy/paste the code for labs, and adjust it to suit the assignments.

- Modifying `StudentRecord.calculate()`

```
    def calculate(self):
        """
        Purpose:
            Calculate the final grade in the course.
        Return:
            The final grade.
        """
        return round(self.midterm.contribution()
                     + self.final.contribution()
                     + sum([b.contribution() for b in self.labs])
                     + sum([a.contribution() for a in self.assignments]))
```

This calculation could have been done by a couple of loops as well.

- Calculating the class average.

```
scores = [a.calculate() for a in course]
print("class average:", sum(scores) / len(scores))
```

A loop or any other method of grabbing all the grades from the student list is fine.

Notes to markers:

- 1 mark: You correctly added the attribute `final_exam` to `StudentRecord`
  - Adding GradeItem for the final is straight-forward. Only deduct the mark if something seriously wrong was done.
- 1 mark: You correctly added the attribute `assignments` to `StudentRecord`
  - Adding a list of GradeItems for the assignments is straight-forward. Only deduct the mark if something seriously wrong was done.
- 4 marks: You correctly modified `StudentRecord.display()` correctly to display the new grade items.
  - Modifying the display is straight-forward. A loop is needed for the assignment GradeItems. Students will probably imitate the code for the lab GradeItems.
  - Deduct 4 marks if something seriously wrong was done.
  - Deduct 2 marks for a minor problem.
- 4 marks: You correctly modified `StudentRecord.calculate()` correctly to calculate the course grade using the new grade items.
  - The main thing here is to bring the new GradeItems into the calculation. This can be done by imitating the way the given version works, and extending it.
  - Deduct 4 marks if something seriously wrong was done.
  - Deduct 2 marks for a minor problem.
- 5 marks: You added some code to the script that calculates the class average.
  - Calculating the class average is straight-forward.
  - Deduct 4 marks if something seriously wrong was done.
  - Deduct 2 marks for a minor problem.