UNIVERSITY OF SASKATCHEWAN

Department of Computer Science
176 Thorvaldson Building
110 Science Place, Saskatoon, SK, S7N 5C9, Canada
Telephine: (306) 966-4886, Facimile: (306) 966-4884

Assignment 4

ADTs and Objects

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

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- This assignment is homework assigned to students and will be graded. This assignment shall not be distributed, in whole or in part, 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 assignment 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.**
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions. Plagiarism can include: copying answers from a web page, or from a classmate, or from solutions published in previous semesters. Basically, if you cannot delete your whole assignment and do it again yourself (given adequate time), it's not your work, so don't try to claim credit for it. Your success in this course depends on what you can do, not on what you can hand in.
- Each question indicates what to hand in. You must give your document the name we prescribe for each question, usually in the form aNqM, meaning Assignment N, Question M.
- Read the purpose of each question. Read the Evaluation section of each question.
- Make sure your name and student number appear at the top of every document you hand in. These conventions assist the markers in their work. Failure to follow these conventions will result in needless effort by the markers, and a deduction of grades for you.
- Do not submit folders, or zip files, even if you think it will help. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.
- **Assignments must be submitted to Moodle.** If you are not sure, talk to a Lab TA about how to do this.
- **Moodle will not let you submit work after the assignment deadline.** It is advisable to hand in each answer that you are happy with as you go. You can always revise and resubmit as many times as you like before the deadline; only your most recent submission will be graded. **Do not send late assignment submissions to your instructors, lab TAs, or markers. If you require an extension, request it in advance using email.**

Version History

- **06/10/2021**: released to students

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 5 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.

Addendum on floating point computations

The following material is a simplified view of floating point numbers. I've taken care to get the principles described correctly, but there are a few details that I have glossed over, which does affect the lessons contained here. You can learn lots more about this matter in CMPT 214, CMPT 215, and even a few MATH courses. This simplified view will be enough for us right now.

Floating point values use a finite number of binary digits ("bits"). For example, in Python, floating point numbers use 64 bits in total. Values that require fewer than 64 bits to represent are represented exactly. Values that require more than 64 bits to represent are limited to 64 bits exactly, but truncating the least significant bits (the very far right).

You are familiar with numbers with infinitely many decimal digits: like π, or 1/3, for example. You may be comfortable with the idea of using 3.14 and 0.333 (or some other number of 3s) as an approximation to the values π and 1/3, especially when doing calculations by hand. Inside a modern computer, floating point numbers are represented in binary as a sequence of exactly 64 binary digits, or "bits." Because of the 64-bit limit, every floating point value like π or 1/3, is cut off (truncated, not rounded) at 64 bits, which corresponds to about 15 decimal digits.

To make this concept a little more precise, consider the value 1/10. We'd normally write 0.1 as a decimal representation of this value. You might realize that there is an infinite sequence of zeroes following the digit 1, even though we don't normally write them. Floating point values, stored inside a computer, use as many zeroes as necessary to fill up the 64-bits. These are always there, even if they are not displayed when printed to the console.

One of the counter-intuitive consequences of the finite floating point representation in binary is that numbers we normally think of as finite when represented in decimal are infinite when represented in binary. For example, the decimal value 0.1, has an infinitely repeating binary representation, rather like the value 1/3 in decimal. The point is that our intuitions, established by years of working with numbers in decimal, can be confused because the computer uses binary.

We're finally getting to the important. We have understood that all floating point values are truncated to fit the 64-bit limit. This leads to an important consequence: **All floating point calculations lead to an accumulation of small errors with every operation.**

The errors that come from use of floating point are unavoidable; these errors are inherent in the accepted standard methods for storing data in computers. This is not weakness of Python; the same errors are inherent in all modern computers, and all programming languages. This is a consequence of modern computing, not Python.

To mitigate the fact that errors always occur, we have to learn the difference between *equal*, and *close enough*, when dealing with floating point numbers. Here are some principles that you should memorize:

- A floating point literal is always equal to itself. In other words, there is no randomness in truncating a long fraction; the following script will display `Equal` on the console, every time, and no matter what floating point literal you use.

```
if 0.1 == 0.1:
    print('Equal')
else:
    print('Not equal')
```

- An arithmetic expression involving floating point numbers is equal to itself. In other words, there is no randomness in errors resulting from arithmetic operations; the following script will display `Equal` on the console.

```
if 0.1 + 0.2 + 0.3 == 0.1 + 0.2 + 0.3:
    print('Equal')
else:
    print('Not equal')
```

If we repeat the exact same operations in the exact same order, we get the exact same result. The error that accumulates is small, but not random.

- If two expressions involving floating point arithmetic are different, the results may not be equal, even if, in principle, they *should be* equal. In other words, errors resulting from floating point arithmetic accumulate differently in different expressions. The following script will display `Not equal` on the console.

```
if 0.1 + 0.1 + 0.1 == 0.3:
    print('Equal')
else:
    print('Not equal')
```

Two algorithms to calculate the same floating point value may result in two values that are close in value, but the errors may accumulate differently. Some sequences of operations can lead to more errors than other sequences. The exact nature of this problem is beyond the scope of this course, though.

- Python's `print()` function will try to simplify or round floating point values. You can tell `print()` to print the number without any simplification, but the number, displayed on the console in decimal can only be an approximation to the 64-bit in the computer, which is also just an approximation to the number your algorithm would calculate in principle.

As a result of the error that accumulates in floating point arithmetic, we have to expect a tiny amount of error in every calculation involving floating point data. **We should almost never ask if two floating point numbers are equal**. Instead we should ask if two floating point numbers are *close enough* to be considered equal, for the purposes at hand.

The easiest way to say *close enough* is to compare floating point values by looking at the absolute value of their difference:

```
# set up a known error
calculated = 0.1 + 0.1 + 0.1
expected = 0.3

# now check for exactly equal
if calculated == expected:
   print('Exactly equal')
else:
   print('Not exactly equal')

# now compare absolute difference to a pretty small number
if abs(calculated - expected) < 0.000001:
   print('Close enough')
else:
   print('Not close enough')
```

The Python function `abs()` takes a numeric value, and returns the value's absolute value. The absolute value of a difference tells us how different two values are without caring which one is bigger. If the absolute difference is less than a well-chosen small number (here we used 0.000001), then we can say it's close enough.

In your test script for this question, you can check if the ADT calculates the right answer by checking if its answer is close enough to the expected value. If it's not, there's a problem!

Question 2 (15 points):

Learning Objectives

Purpose: Students will practice the following skills:

- Reading and underdtanding 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 ~~three~~ 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.

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.