Assignment 3

Software Testing and ADT

Date Due: 31 January 2020, 8pm **Total Marks: 60**

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.
- **Assignments are being checked for plagiarism.** We are using state-of-the-art software to compare every pair of student submissions.
- 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.
- 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.** There is a link on the course webpage that shows you 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.
- Read the purpose of each question. Read the Evaluation section of each question.

Version History

- **22/01/2020**: Release to students.

Overview

In this assignment, you'll be finding and fixing bugs in a couple of scripts we provide for you (first two questions). The last question will require you to complete a test script for an ADT.

To theme of the question 1 and 2 is a popular game called Connect Four. In the commercial version of the game, the board is limited to 5×7, but the scripts allow various size boards. The game requires players to take turns placing a token in the board. To make the description clear, we will refer to player 1 as the player who uses the token `'X'`, and player 2 as the player who uses the token `'O'`. A player wins if they can get four of their own tokens sequentially in any row, column or diagonal. This is similar to TicTacToe, except that in Connect Four we have "gravity", which means that a player can choose which column to place a token, but the row is chosen as the lowest unoccupied position in that column.

We will not be writing the game-playing part of the program, as that's a bit too advanced for first year. A task we can do is to check if there are winning sequences for one player or another. This exercise is very similar to some of the work you did in Assignment 1. Another task is to write a script that produces randomly generated boards, to provide data for our board checking script.

In the questions below, you are given two scripts which do not yet work. Your job will be to test and debug the scripts, and submit the test scripts and the working code. You should review Chapters 2 and 5 in the course readings, and apply the concepts described there.

You can modify the code in many ways; there is no one right answer. But you should resist the urge to start from scratch and solve the problem yourself. Working with code that is not your own is sometimes uncomfortable, but it is a reality in industry, and a reality in the life of any programmer. While it may seem hard to believe, when a programmer returns to a script they wrote as little as a few months previously, it feels like someone else wrote the program.

Question 1 (20 points):

Purpose: To practice testing and debugging, and working with unfamiliar code.

Degree of Difficulty: Tricky. This will take some time!

You will find a Python script named `randomc4.py` on the Assignment 3 Moodle page. Its intended purpose is to display on the console a number of random-generated Connect-4 "boards." As an example of correct behaviour, here's a correct "board":

```
2
6 8
XO......
XOX.....
OOO..OXX
XXX..XXO
XOX.OXOO
XOXOXOOO


5 9
.........
.........
.........
.OOXO....
OOXXX..X.
```

The first line of the output indicates how many boards are being displayed; in this case, there are 2. After that, the output tells the height (6) and width (8) of the first board. After that, there are 6 lines of text (the number of lines depends on the indicated height), with one character per board position. There are Xs and Os, and the character '.' which means it's unoccupied. The script should display several such boards, of various random sizes, and a blank line always separates an example from any board that follows. There is a second board in the above example, with its own height and width, and distribution of tokens.

This script was written and tested by an expert Python programmer, and all tests were passed. There are several example files on Moodle that show correct output from a correct version of this script. Because it generates randomized boards, you won't see the exact output that we have shown. In this question, we are not concerned about checking if the board is valid or not. The boards you generate may have one or more sequences of either token; it's random, and doesn't have to be perfectly realistic.

The script we provide to you started with the correct script, but then modified by adding bugs into the script, similar in nature to the kinds of bugs that plague first year students, and very much related to the ideas of Chapter 2 of the course readings, and Lab 02.

Your task is:

1. Read and familiarize yourself with the code.
2. Design and implement test cases, running them, and using their output as evidence.
3. Use debugging skills, including the PyCharm debugger!
4. Fix bugs and get the script working again.

What to Hand In

- A test script named `a3q1_testing.py`
- The fixed file `randomc4.py`

- A text file named `a3q1.txt` with a brief (1-3 sentences) description of each problem you fixed.

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

Evaluation

- 8 marks: Your test script did a thorough job of testing the script.
- 4 marks: Your version of `randomc4.py` produces correct output.
- 8 marks: Your description file identifies most of the problems introduced into script.

Question 2 (30 points):

Purpose: To practice testing and debugging, and working with unfamiliar code.

Degree of Difficulty: Tricky. This will take some time!

You will find a Python script named `checkc4.py` on the Assignment 3 Moodle page. Its intended purpose is to read a named file, containing examples of Connect Four boards, as described in the previous question (in fact, these files were generated by a correctly implemented version of A3Q1). For each board in the file, the script will display one of three messages to the console:

- `X`: indicating that there are 4 `X` tokens in a sequence in some row, column or diagonal. This is like saying "Player 1 wins!"
- `O`: indicating that there are 4 `O` tokens in a sequence in some row, column or diagonal. This is like saying "Player 2 wins!"
- `No decision` indicating that there were no sequences of 4 or more tokens for either player.

Note: Some random examples may have more than 4 consecutive tokens; the sequences were generated randomly. That's okay; we'll count it as a win for that player. Also, there may be sequences of 4 tokens for both players in the board; in this case, you can output either token.

Like A3Q1, the script was written and tested by an expert Python programmer, and all tests were passed. But then bugs were introduced into the script, similar in nature to the kinds of bugs that plague first year students, and very much related to the ideas of Chapter 2 of the course readings, and Lab 02.

Your task is:

1. Read and familiarize yourself with the code.
2. Design and implement test cases, running them, and using their output as evidence.
3. Use debugging skills, including the PyCharm debugger!
4. Fix bugs and get the script working again.

There are some example files on the Assignment 1 Moodle page which you can use as input, and if you got A3Q1 working, you can generate as many examples as you need.

What to Hand In

- A test script named `a3q2_testing.py`
- The fixed file `checkc4.py`
- A text file named `a3q2.txt` with a brief (1-3 sentences) description of each problem you fixed.

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

Evaluation

- 12 marks: Your test script did a thorough job of testing the script.
- 6 marks: Your version of `checkc4.py` produces correct output.
- 12 marks: Your description file identifies most of the problems introduced into script.

Question 3 (10 points):

Purpose: Completing a test script for an ADT.

Degree of Difficulty: Easy.

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, and sometimes the tests look into the ADT's data structure, and sometimes, the operations are used to help set up tests. You'll notice that there is exactly one test for each operation, which is inadequate.

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 ***'`.

What to Hand In

- A Python script named `a3q3_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

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: π, for example, is often cut off at 3.14 when calculating by hand. Inside a modern computer, π is limited to about 15 decimal digits, which fits nicely in 64 bits. Because long fractions are truncated, many calculations inside the computer are performed with numbers that have been truncated, leading to an accumulation of small errors with every operation.

Another value with an infinite decimal fraction is the value 1/3. But because a computer uses binary numbers, some values we think of naturally as "finite" are actually infinite. For example, 1/10 has a finite decimal representation (0.1) but an infinite binary representation.

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.

We have to learn the difference between *equal*, and *close enough*, when dealing with floating point numbers.

- 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.

```
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 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')
```

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!