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: 15 February 2023, 11:59pm **Total Marks: 39**

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 Canvas.** There is a link on the course webpage that shows you how to do this.
- **Canvas 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.

Question 1 (5 points):

Purpose: To force the use of Version Control in Assignment 4

Degree of Difficulty: Easy

Version Control is a tool that we want you to become comfortable using in the future, so we'll require you to use it in simple ways first, even if those uses don't seem very useful. Do the following steps.

1. Create a new PyCharm project for Assignment 4.
2. Use `Enable Version Control Integration...` to **initialize** Git for your project.
3. Download the Python and text files provided for you with the Assignment, and **add** them to your project.
4. Before you do any coding or start any other questions, make an initial **commit**.
5. As you work on Assignment 4, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed the question, or want to try something different. Make your most professional attempt to use the software appropriately.
6. When you are finished your assignment, open PyCharm's Terminal in your Assignment 4 project folder, and enter the command:

   ```
   git --no-pager log
   ```

   WARNING: Copy/Paste from PDFs can often add unwanted spaces. There are no spaces between dashes (-) in the above command!
7. Copy/Paste the output of the above command into a text file named `a4-git.txt`.

Notes:

- Several Version Control videos are available on Canvas via the Lecture Videos link.
- No excuses. If you system does not have Git, or if you can't print the log as required, you will not get these marks. Installation of Git on Windows 10 machines is outlined in the "*Step-by-Step Windows 10 - PyCharm UNIX command-line*" video. Git is included in base installations of Linux & MacOS.
- If you are not using PyCharm as your IDE, you are still required to use Git. See the Version Control videos on Canvas that cover using Git on the command line & via GUI.

What Makes a Good Commit Message?

A good commit message should have a **subject line** that describes *what* changed, a **body** that explains *why* the change was made, and any other *relevant* information. Read "*Writing a Good Commit Message*" linked on Canvas for more guidance.

Examples of GOOD commit messages:

```
Initial commit for Assignment 4

* copied starter files over from Canvas
```

```
Completed Question 2

* full functionality complete after testing with generated files
```

```
Fixed a bug causing duplicate print messages

* extra print statements were being produce in loop
* updated logical condition to fix
```

Examples of BAD commit messages:

```
fixes
```

```
add tests
```

```
updates
```

What to Hand In

After completing and submitting your work for the Assignment 4, **follow steps 6 & 7 above** to create a text file named `a4-git.txt`. Be sure to include your name, NSID, student number, course number, and lecture section at the top of your `a4-git.txt` file.

Evaluation

- 5 marks: The log file shows that you used Git as part of your work for Assignment 4.

  For full marks, your log file contains
  - Meaningful commit messages.
  - A minimum of 4 separate commits.

Question 2 (10 points):

Purpose: Completing a test script for an ADT.

Degree of Difficulty: Easy.

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 Canvas, you'll find the following file(s):

- `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.
- `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:

- White-box test cases.
- Black-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 `a4q2_testing.py` containing your test script.

Be sure to include your name, NSID, student number, course number and 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!

Question 3 (12 points):

Purpose: Adding operations to an existing ADT; completing a test script

Degree of Difficulty: Moderate.

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 Canvas, you'll find the following file(s):

- `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 four new operations for the ADT:

- `range()` Returns the range (difference between the minimum and maximum) based on every value recorded by the Statistics object.
- `mode()` Returns the mode (the value that occurs most frequently) based on every value recorded by the Statistics object.
- `max()` Returns the maximum value ever recorded by the Statistics object.
- `min()` Returns the minimum value ever recorded by the Statistics object.

Note: For each of the above, if not data was seen, the function should return `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 Q2 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 `a4q3_changes.txt` describes changes you made to the existing ADT operations.
- A Python program named `a4q3.py` containing the ADT operations (new and modified).
- A Python script named `a4q3_testing.py` containing your test script.

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

Evaluation

- 1 mark: Your added operation `range()` is correct and documented.
- 1 mark: Your added operation `mode()` is correct and documented.
- 1 mark: Your added operation `max()` is correct and documented.
- 1 mark: Your added operation `min()` is correct and documented.
- 3 marks: Your modifications to other operations are correct.
- 5 marks: Your test script has good coverage for the new operations.

Question 4 (12 points):

Purpose: To implement an ADT from a requirements specification.

Degree of Difficulty: Moderate

A pack of playing cards has four suits, 'H', 'D', 'S', and 'C', (Hearts, Diamonds, Spades and Clubs) and a value 1 through 13 making a total of 52 cards. The following symbols map to the following values: A=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 10=10, J=11, Q=12, K=13 (with A being the Ace, J being the Jack, Q being the Queen and K being the King). The suit a card belongs to does not affect its value, so, 2H and 2S both have the value of 2. On the other hand, QH has a higher value of 12 compared to AH that has a value of 1.

You are required to design and implement an ADT named card. A card is a data structure that carries out some operations on a pack of cards. Each card itself will be stored as a string, with a numeric value preceding a letter (ex: "JD" for the jack of diamonds, "5H" being the 5 of hearts, etc).

The card ADT has 6 operations:

- `Card.create()` Creates and returns a list of all possible cards that can be drawn from the 52 card playing deck. That is, all possible combinations of the suits of cards ['H', 'D', 'S', 'C'] and the values 1 through 13 (but using A as 1, J as 11, Q as 12, K as 13).

```
print( Card.create() )
# ["AH","2H","3H","4H","5H","6H","7H","8H","9H","10H","JH","QH","KH",...]
# Repeated for the 3 other suits (D,S,C)
```

- `Card.deal(num_cards, num_players, deck)` Randomly deals a number of cards to a number of players from the given deck (list of card strings). When a card is drawn, it is removed from the deck. deal() returns each player's hand as a list of lists. It is up to you to decide what happens if the deck runs out of cards before all players have been dealt all of their cards.

```
cards_dealt = Card.deal(5, 3, deck)
print(cards_dealt)
# [["5D","10H","QS","8C","JH"],
#  ["JC","2H","8D","AS","9C"],
#  ["7H","6H","9H","10S","8H"]].
```

- `Card.value(card)` Takes in the string of a card, and returns its integer value. Remember, its suit does not affect its value. Also remember that A=1, J=11, Q=12, K=13. For example, Card.value("KS") returns 13, and Card.value("4D") returns 4.

- `Card.highest(list_of_cards)` Takes a list of cards, and returns the the card string with the highest value. For example:

```
highest_card = Card.highest(["5D","10H","QS","8C","JH"])
print(highest_card) # outputs: "QS"
```

- `Card.lowest(list_of_cards)` Takes a list of cards, and returns the string with the lowest value. For example:

```
lowest_card = Card.lowest(["5D","10H","QS","8C","JH"])
print(lowest_card) # outputs: "5D"
```

- `Card.average(list_of_cards)` Takes a list of cards, and returns the average (float) value of this given list of cards.

```
average = Card.average(["5D","10H","QS","8C","JH"])
# Will compute the average for: 5, 10, 12, 8, 11
print(average) # outputs: 9.2
```

Testing

Write a test script that tests your implementation of the ADT, and ensures that all operations are correct. Think about the same kinds of issues:

- White-box test cases.
- Black-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 your correct ADT should report no errors, and should display nothing except the message `'*** Test script completed ***'`.

What to Hand In

- A Python program named `a4q4.py` containing the ADT operations (new and modified).
- A Python script named `a4q4_testing.py` containing your test script.

Evaluation

- 1 mark: Your operation `create()` is correct and well documented.
- 1 mark: Your operation `value(card)` is correct and well documented.
- 1 mark: Your operation `deal(num_cards, num_players, deck)` is correct and well documented.
- 1 mark: Your operation `highest(list_of_cards)` is correct and well documented.
- 1 mark: Your operation `lowest(list_of_cards)` is correct and well documented.
- 1 mark: Your operation `average(list_of_cards)` is correct and well documented.
- 6 marks: Your test script checks that the operations work correctly.