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 3

ADTs and Testing

Date Due: Feb 2, 2017, 10pm **Total Marks: 52**

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.
- 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.
- 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

- **28/01/2018**: Revised the example output for Q4 running on the concsole with the given example file. Your program should count `','` and `'!'` as normal characters.
- **28/01/2018**: Revised and extended the discussion of floating point errors. See Q1!
- **27/01/2018**: Added a discussion of floating point errors. See Q1!
- **24/01/2018**: released to students

Question 0 (10 points):

Purpose: To require the use of Version Control in Assignment 3

Degree of Difficulty: Easy

You are expected to practice using Version Control for Assignment 3. This is a tool that you need to be required to use, even when you don't need it, so that when you do need it, you are already familiar with it. Do the following steps.

1. Create a new PyCharm project for Assignment 3.
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. Use `PyCharm > VCS > Git > Add`. It's not enough to drop them into your project.
4. Before you do any coding or start any other questions, make an initial version (commit).
5. As you work on each question, use Version Control frequently at various times when you have implemented an initial design, fixed a bug, completed a 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 the terminal in your Assignment 3 project folder, and enter the command: `git --no-pager log` (double dash before the word 'no'). The easiest way to do this is to use PyCharm: locate PyCharm's `Terminal` panel at the bottom of the PyCharm window, and type your command-line work there.
   Note: You might have trouble with this if you are using Windows. Hopefully you are using the department's network filesystem to store your files. If so, you can log into a non-Windows computer (Linux or Mac) and do this. Just open a command-line, `cd` to your A3 folder, and run `git --no-pager log` there. If you did all your work in this folder, Git will be able to see it even if you did your work on Windows. Git's information is out of sight, but saved in your folder.
   Note: If you are working at home on Windows, Google for how to make git available on your command-prompt window. You basically have to tell the command-line app where the git app is.

You may need to work in the lab for this; Git is installed there. Not having Git installed is not an excuse. It's like driving a car without wearing a seatbelt. It's not an excuse to say "My car doesn't have a seatbelt."

What to Hand In

After completing and submitting your work for Questions 1-4, open a command-line window in your Assignment 3 project folder. Run the following command in the terminal: `git --no-pager log` (double dash before the word 'no'). Git will output the full contents of your interactions with Git in the console. Copy/-paste this into a text file named `a3-git.log`.

If you are working on several different computers, you may copy/paste output from all of them, and submit them as a single file. It's not the way to use Git, but it is the way students work on assignments.

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

Evaluation

- 10 marks: The log file shows that you used Git as part of your work for Assignment 3. For full marks, your log file contains
  - Meaningful commit messages.
  - At least three commits per question for a total of at least 12 commits. And frankly, if you only have 12 commits, you're pretending. But at least you're practicing.

Question 1 (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 `a3q1_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!

Question 2 (11 points):

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

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.

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

- `count(stat)` Returns the number of data values that the Statistics record `stat` has already recorded.
- `maximum(stat)` Returns the maximum value ever recorded by the Statistics record `stat`. If no data was seen, returns `None`.
- `minimum(stat)` Returns the minimum value ever recorded by the Statistics record `stat`. 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 `a3q2_changes.txt` (other acceptable formats) describes changes you made to the existing ADT operations. Be brief!
- A Python program named `a3q2.py` containing the ADT operations (new and modified), but no other code.
- A Python script named `a3q2_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

- 1 mark: Your added operation `count(stat)` is correct.
- 1 mark: Your added operation `maximum(stat)` correct.
- 1 mark: Your added operation `minimum(stat)` correct.
- 3 marks: Your modifications to other operations are correct.
- 5 marks: Your test script has good coverage for the new operations.

Question 3 (11 points):

Purpose: To implement an ADT from a requirements specification.

Degree of Difficulty: Moderate

Design and implement an ADT named `Counter`. A counter is a data structure that records how often data values are observed. For example, you might want to use a counter to record the frequency of letters in a string, or the frequency of words in a file. The counter ADT provides the operations to make the counting easier. We will assume that we'll be counting values like integers and strings, which are immutable.

A counter has four operations:

- `Counter.create()` Creates a data structure to record the number of (immutable) data values are seen. A reference to the new counter is returned.
- `Counter.see(counter, value)` records the observation of the given `value` in the `counter`.
- `Counter.seen(counter, value)` reports how many times the given `value` has been seen before, as recorded in the `counter`.
- `Counter.total(counter)` reports how many values have been seen before in total, as recorded in the `counter`.
- `Counter.size(counter)` reports how many unique values have been seen, as recorded in the `counter`.
- `Counter.unique(counter)` returns a list of all the unique values that have been seen, as recorded in the `counter`.

To get you started, we have provided a starter file, with some documentation, and operations that are no more than stubs. Fill in the implementations.

Testing

Write a test script that tests your implementation of the ADT, and ensures that all operations are correct. Use the example from Question 1 as your basis. Think about the same kinds of issues:

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

Hint: You might start out by writing a test script based on your understanding of what the operations should do (black-box). If you run the test script right away, you'll get a lot of errors (because the starter file doesn't do anything useful yet). Then implement each function one at a time, debugging and correcting before you move on. Add a few tests when you're done to be sure that you haven't settled on an implementation that is too closely matched to your initial set of tests.

What to Hand In

- A Python program named `a3q3.py` containing the ADT operations (new and modified), but no other code.
- 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

- 1 mark: Your operation `create()` is correct.
- 1 mark: Your operation `see(counter, value)` is correct.
- 1 mark: Your operation `seen(counter, value)` is correct.
- 1 mark: Your operation `size(counter)` is correct.
- 1 mark: Your operation `total(counter)` is correct.
- 1 mark: Your operation `unique(counter)` is correct.
- 5 marks: Your test script checks that the operations work correctly.

Question 4 (10 points):

Purpose: To practice using an ADT you built yourself in a relatively simple application.

Degree of Difficulty: Easy

Write a Python program that reads a named text-file from the current working directory (e.g., a file with `.txt` or `.py` extension) and displays to the console the characters (letter, number, or symbol) used in the file, along with the number of times it appears in the file.

Your program should run from the command-line, and use a file-name that was given on the command-line. For example, suppose the file `hello.txt` contains the following two lines of text:

```
UNIX$ cat hello.txt
Hello,
world!
```

Running your program on the command line would produce the following results:

```
UNIX$ python3.6 a7q1.py hello.txt
l : 3
e : 1
, : 1
H : 1
o : 2
r : 1
d : 1
! : 1
w : 1
UNIX$
```

The order that the characters appear in the output is not important. The Moodle page for this assignment includes a number of text files for your demonstrations.

What to Hand In

- Hand in a well-written, suitably documented program named `a3q4.py`.
- Run your program on a few short files of your own. Copy/paste the command-line interaction, as in the above example, showing the contents of your files using the UNIX command `cat` and your program. Consider this a demonstration, not testing. Because this application is pretty simple, and because you tested your Counter ADT very thoroughly, you only need to rune a few demonstrations of your program working.

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

Evaluation

- 5 marks: Your program makes appropriate use of the Counter ADT, and is otherwise correct.
- 5 marks: Your demonstration shows your program working. If we run your program on the command-line, we'll see the same results.