Assignment 9

Testing and Debugging

Date Due: Thursday, March 25, 11:59pm **Total Marks: 21**

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 (yes, sometimes typos happen – do not be confused). 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.** Marks will be deducted with severity if you submit code for Python 2.
- **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.
- Questions are annotated use descriptors like "easy" "moderate" and "tricky". All students should be able to obtain perfect grades on "easy" problems. Most students should obtain perfect grades on "moderate" problems. The problems marked "tricky" may require significantly more time, and only the top students should expect to get perfect grades on these. We use these annotations to help students be aware of the differences, and also to help students allocate their time wisely. Partial credit will be given as appropriate, so hand in all attempts.
- Read the purpose of each question. Read the Evaluation section of each question.

Question 1 (7 points):

Purpose: To practice using blackbox testing without seeing a function's code

Degree of Difficulty: Easy

Black box tests are typically best written BEFORE you even start writing the function they are intended to test. For this question, you will write a series of black box tests for a function that you never write or see yourself.

The recentTrend() function

The function you are testing is called `recentTrend()`. The function accepts a single parameter, which is a **list of integers**. The function should return the **most common value** among the **last 20 elements** of the list (i.e. the 20 values at the END of the list). Some special cases for the return value:

- If the list is empty, the value `None` should be returned
- If the list has fewer than 20 elements, it should just return the most common value
- If there is a tie for the most common value, than the **largest value** from among those tied should be returned

Remember: **you are not writing this function**. You just need to understand what it is **supposed** to do so that you can write effective test cases for it.

Write a test driver

Write a test driver that contains several tests for the `recentTrend()` function. A starter file is provided that contains an example of a single test case. Add additional tests to this file, following the example of section 15.2.4 of the textbook.

Choose your test cases thoughtfully to cover a range of possible situations. Do NOT bother with test cases that use incorrect data types (i.e. passing in something other than a list, or a list of booleans instead of a list of integers). Instead, focus on tests to expose any possible errors in the function's logic. Exactly how many test cases to use is up to you; include as many as you think you need to discover any errors in the function.

What to Hand In

- A document called `a9q1_testdriver.py` containing your test driver implementation.

Be sure to include your name, NSID, student number and instructor's name at the top of all documents.

Evaluation

- 3 marks: The **form** of the test driver (calling the function, comparing the result to a correct expected result) is correct
- 3 marks: The **quality** of the selected tests is good (they cover a range of cases and show evidence of thoughtful design)
- 1 mark: Your tests discover an intentional error that has been placed in `recentTrend()`
- **-1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file.**

Question 2 (8 points):

Purpose: To create white-box tests for a function you wrote yourself.

Degree of Difficulty: Moderate, only because some students struggle with white-box test reasoning

White-box tests need to be created after the function being tested has been written. The idea behind white-box testing is that perhaps looking at/writing the code will give you an idea for a test case that you may not have thought of otherwise. For this question, you'll create some white-box tests to test a function that you wrote yourself.

The closest_to_zero() function

Write a function called `closest_to_zero(num1, num2, num3)`. The function should have **3 parameters which are each integers**, and returns the value that is **closest to 0** from among those 3.

For example, given the inputs `2`, `7` and `0`, the function should return `0`. Given the inputs `3`, `-1` and `5`, the function should return `-1`.

It is possible that there is a tie. For example, `1` and `-1` are equally close to 0. In the case of a tie between a positive and negative value, the function should return the positive value.

Hint: the `abs()` function may be useful in your solution.

Write a test driver

Write a test driver that contains several tests for the `closest_to_zero()` function. Use the example of section 15.2.4 of the textbook to get an idea for the format of the tests.

Choose your test cases thoughtfully, using insights gained from writing the code. Think about every single **individual line** of code that you wrote, and how you might create a test case to test that line. Do NOT bother with test cases that use incorrect data types (i.e. passing in strings instead of integers). Exactly how many test cases to use is up to you; include as many as you think you need to be confident that your function is correct.

What to Hand In

- A document called `a9q2.py` containing your function.
- A document called `a9q2_testdriver.py` containing your test driver. This file can `import` your function from `a9q2.py` in order to test it.

Be sure to include your name, NSID, student number and instructor's name at the top of all documents.

Evaluation

- 2 marks: Your function `closest_to_zero()` takes three integers and correctly finds the integer closest to zero.
- 3 marks: The **form** of the test driver (calling the function, comparing the result to a correct expected result) is correct
- 3 marks: The **quality** of the selected tests is good (they cover a range of cases and show evidence of **white-box** thinking and design)
- **-1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file.**

Question 3 (6 points):

Purpose: Practice tracing code to help debugging.

Degree of Difficulty: Easy

Once we have identified faults by testing, we need to find the causes of faults and correct the code by debugging. Hand-tracing a function's execution is one way to do this. In this question, you will practice tracing a function called `hasMajority()` line by line.

The function is provided on the class website, along with a test function call. Here is a description of WHAT the function is supposed to do:

- `hasMajority()`: Given a list of size N, returns `True` if the list has a **majority element**, and `False` otherwise
- **majority element**: A majority element is an element that appears more than `N//2` times in the list (i.e. a single repeated element that makes up more than half the contents of the list)

The code is included below. Briefly, the function first builds a dictionary where keys are data elements from the input list and values are the count of that element (effectively, this is a data structure called a **histogram**). Then the function checks whether the element with the highest count in the dictionary is a majority element according to the definition above.

```
1  def hasMajority(ls):
2      '''
3      Determines if the input list "ls" includes the majority element or not.
4      :param ls: an arbitrary list with comparable elements
5      :return: True if the input list has a majority element; False otherwise.
6      '''
7      # initialize the dictionary with keys are the elements in ls,
8      # and values are the count the key in ls.
9      counts = {}
10     # update the dictionary of counts by loop in the list
11     for i in ls:
12         if i in counts:
13             counts[i] += 1
14         else:
15             counts[i] = 0
16     # generate the max count of all values in the list
17     max_count = max(counts.values())
18     # generate the result
19     result = max_count > len(ls)//2
20     return result
21
22 isMajorityList([1,2,3,1,1])
```

Your task:

- Fill in the table below by tracing the change of variables in the function line by line. You can use either hand-tracing or pycharm's integrated debugger. The first line (Line 9) is already done for you as an example. For the document you actually hand in, a spreadsheet program like Excel works well to create the table.
- Find the error. There is a small **error** in this function; it can be fixed by changing JUST ONE line of code. Using the insight gained from making your table, determine which line contains the error and indicate how can it be corrected. You don't need to hand in the fixed code, just include a sentence after the table in the document to indicate where the error is and how to fix it.

line - iteration #	`counts`	`i`	`i in counts`	`max_count`	`result`
9	`{}`	`undefined`	`undefined`	`undefined`	`undefined`
11 - iteration 1					
12 - iteration 1					
13 - iteration 1					
15 - iteration 1					
11 - iteration 2					
12 - iteration 2					
13 - iteration 2					
15 - iteration 2					
11 - iteration 3					
12 - iteration 3					
13 - iteration 3					
15 - iteration 3					
11 - iteration 4					
12 - iteration 4					
13 - iteration 4					
15 - iteration 4					
11 - iteration 5					
12 - iteration 5					
13 - iteration 5					
15 - iteration 5					
17					
19					

What to Hand In

- A document called `a9q3.pdf` (rtf and txt formats are also allowable) with the table you filled, and the explanations of the errors you found. A program like Excel or else browser-solutions like Google Sheets work well to create this table, and have an option to easily export to pdf.

Evaluation

- 1 marks: A table with a reasonable format is submitted that shows a full trace
- 4 marks: The values in the trace are correct for each line
- 1 marks: You correctly identified the error and explained briefly how you fixed it.
- **-1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file.**