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 6

Software Design and Testing

Date Due: 5:00pm Friday Nov 1 **Total Marks: 25**

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. It might help you, but it adds an extra step for the markers.
- Programs must be written in Python 3.5+
- You may not, in part or in whole, submit any code that was written by generative AI tools, such as ChatGPT
- **Assignments must be submitted to Canvas (the course website).**
- **Canvas will not let you submit work after the assignment hand-in closes.** 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 practice creating dictionaries in a list of records

Pokemon are fantastic creatures that people can catch and train. Information about pokemon is typically stored in a digital document known as a Pokedex.

Combining lists with dictionaries as a **list of records** is a very useful way to store data. For this question, you'll practice this technique by using a list of records to log and store the data for a user's Pokedex.

Function to make a Record

There are three pieces of information we'll need to store about each pokemon: its species name, its elemental type, and its combat level. For example, perhaps the user will wish to log a level 33 Pikachu (the species), which is an Electric-type pokemon. We will use a dictionary as a record to store the information for each pokemon.

Define a function that accepts a pokemon's species, type and level, and creates a new dicitonary for that pokemon. The dictionary should have the keys `"Species"`, `"Type"`, and `"Level"`, and each key should be mapped to the relevant info about the pokemon. The function should then return this dictionary.

Create a list of records

In the main part of your program, initialize an empty list to represent the PokeDex. This list is ultimately where we'll put all of the pokemon information.

Then write a simple loop that asks the user to keep typing in pokemon info as many times as needed until they are done. For each pokemon, call the function from the previous section to create a pokemon record, and add that record to the PokeDex.

Once the user is done adding pokemon, display the final PokeDex to the console.

Sample Run

Your output might look something like this:

```
Welcome to PokeDex Logger!
Enter info for newly-caught Pokemon
Pokemon's species? pikachu
Pokemon's type? electric
Pokemon's level? 33
----
Are there more pokemon to add (y/n)? y
Enter info for newly-caught Pokemon
Pokemon's species? squirtle
Pokemon's type? water
Pokemon's level? 1
----
Are there more pokemon to add (y/n)? n
Logging complete. Printing final PokeDex.
[{'Species': 'pikachu', 'Type': 'electric', 'Level': 33},
     {'Species': 'squirtle', 'Type': 'water', 'Level': 1}]
```

What to Hand In

- A file called `a6q1.py` containing your finished program, as described above.

Evaluation

- 3 marks for overall correctness
- 2 marks for correct design of record fields and function interface

Question 2 (7 points):

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

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 improvedAverage() function

The function you are testing is called `improvedAverage()`. The function accepts a single parameter, which is a **list of integers**. The function should return the value `True` if the **average** of the **last 10 values** in the list is **greater** than the **average** of all of the **other values** (i.e. everything else EXCEPT the last 10 values), and `False` otherwise.

If the list is empty or contains fewer than 20 values, the value `None` should be returned instead.

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 `improvedAverage()` 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 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 `a6q2_testdriver.py` containing your test driver implementation.

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: Our version of `improvedAverage()` contains an intentional error. 1 mark here if your tests are thorough enough to find it!

Question 3 (8 points):

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

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 examples 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 `a6q3.py` containing your function.
- A document called `a6q3_testdriver.py` containing your test driver. This file can `import` your function from `a6q3.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)

Question 4 (3 points):

All programs that you submit for this assignment will be assessed with regard to their **style and readability**. Make sure that all of your code:

- includes your name, nsid, and student number as comments
- uses informative variable names
- makes effective use of formatting and white space
- includes a `docstring` for all **function definitions** in the program
- for longer programs, uses helpful and concise single-line comments where appropriate

What to Hand In

There is nothing to hand in for this question. It's simply describing how the rest of your work will be evaluated with regard to style and readability. It is possible to earn these marks even if you don't hand in every other question in this assignment, but you must hand in enough work such that your style and readability can be adequately assessed.

Evaluation

- 3 marks for consistently following the principles listed above

Question 5 (2 points):

In your submission for this assignment, you have an opportunity to earn something we'll call "the wow factor". To earn the wow factor, you must do something creative that goes beyond the expectations laid out in the assignment. This could be an extra feature for one of the previous questions, or it could be an extra little program that you submit that builds on the skills used in the assignment.

You should not aim to do a lot of work for this, but should instead demonstrate in a simple and effective way the depth of your understanding. In short, take ownership of your work and "impress us".

Telling you exactly what to do to earn "the wow factor" would defeat the entire purpose, but here are some very general ideas.

- improve the aesthetic appeal of your program(s)
- improve the user-friendliness of your program(s)
- remove a simplifying assumption and handle the resulting added complexity
- compare two different ways of doing the same task, and submitting data/analysis on which was better
- just doing something generally cool

One quick warning: using fancy extra code libraries in order to AVOID a key learning objective from the assignment is not impressive, it's actually anti-impressive. Focus instead on solving problems using simple clean foundational programming skills. Using extra libraries is fine if they let you do something BEYOND what the assignment asked, but not if they're replacing a core assigned task.

What to Hand In

A plain text file called `wow.txt` describing what you did that you think merits the wow factor. One or two sentences is enough. If you create any additional code files, hand those in too.

Evaluation

- 2 marks for something impressive