Assignment 4

Repetition

Date Due: Thursday, October 8, 11:59pm **Total Marks: 26**

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. Put your name and student number 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, zip documents, even if you think it will help.
- Programs must be written in **Python 3**, and the file format must be text-only, with the file extension `.py`.
- Documents submitted for discussion questions should make use of common file formats, such as plain text (`.txt`), Rich Text (`.rtf`), and PDF (`.pdf`). We permit only these formats to ensure that our markers can open your files conveniently.
- **Assignments must be submitted electronically 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 using 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 anything you've done, even if it's not perfect.

Question 1 (7 points):

Purpose: To practice using loops to repeat actions until a condition is met (and solve a practical problem at the same time!).

Degree of Difficulty: Easy

Radioactive materials decay over time and turn into other substances. For example, the most common isotope of uranium, U-238, decays into thorium-234 by emitting an alpha-particle.

The rate of radioactive decay of a material is measured in terms of its **half-life**. The half-life of a material is defined to be the amount of time it takes for half of the material to undergo radioactive decay. Let m be the **initial mass** in grams of some material and let h be the material's **half-life** in days. Then the **remaining mass** of the material on day t, denoted $m(t)$, is given by the formula:

$$m(t) = m \times 0.5^{t/h}$$

Note: This formula is not a Python assignment statement! It is an equation that says: if you know the quantities m, t, and h, you can calculate a value using the right side of the equation. That value will be the amount of **remaining mass** of the material after t days.

For this question, write a program which does the following:

- Prompt the user to enter the **initial mass** of the material (in grams). You must make sure that the user enters a positive number. If they do not enter a positive number, print a message informing them so, and prompt them to enter the initial amount of material again. Keep doing this until they enter a positive number.
- Prompt the user to enter the **half-life** of the material (in days). As above, make sure that the user enters a positive number, and if they don't, keep asking until they do.
- Starting from day 0, output the amount of the material remaining at one-day intervals. Thus, for day 0, day 1, day 2, etc., you should print out the amount of remaining mass according to the above formula. Your program should stop on the first day on which remaining mass is less than 1% of the initial mass.

Don't forget to import the math module if you need math functions.

Hint: A correct solution should make use of three while-loops.

Sample Run

Sample input and output (input typed by the user is shown in green text):

```
Enter half life in days: -7
Error!  Half life must be a positive number.
Enter half life in days: 2
Enter initial amount of material in grams: 100
After 0 days there are 100.0 grams remaining.
After 1 days there are 70.71067811865476 grams remaining.
After 2 days there are 50.0 grams remaining.
After 3 days there are 35.35533905932738 grams remaining.
After 4 days there are 25.0 grams remaining.
After 5 days there are 17.67766952966369 grams remaining.
After 6 days there are 12.5 grams remaining.
After 7 days there are 8.838834764831844 grams remaining.
After 8 days there are 6.25 grams remaining.
After 9 days there are 4.419417382415922 grams remaining.
After 10 days there are 3.125 grams remaining.
After 11 days there are 2.209708691207961 grams remaining.
After 12 days there are 1.5625 grams remaining.
After 13 days there are 1.1048543456039805 grams remaining.
After 14 days there are 0.78125 grams remaining.
```

Note how the half-life in this example was 2 days, and that indeed, after 2 days, half of the original 100 grams of material decayed and we were left with 50 grams. Further note how an after an additional 2 days, on day 4, half of the 50 grams remaining on day 2 decayed leaving us with 25 grams remaining. This observation should help you determine whether your program is correct for other values of h – after every h days, the amount of material should have decreased by half.

Testing

Test your program with $h = 2$ and $m = 100$ and make sure the output is identical to the sample output. Then test your program with $h = 1$ and $m = 100$. Make sure the output makes sense! Then test your program **once more** with inputs of your choice — these should be unique to you and unlikely to be chosen by two different students. Do not use too large a value for h, or your program will produce very long outputs. Copy the console output of your **three test runs** to a document, and hand it in with your program code (see below).

What to Hand In

- A file called `a4q1.py` containing your finished program, as described above.
- A document called `a4q1_testing` that contains your console output. Allowed file formats are plain text (.txt), Rich Text (.rtf) and PDF (.pdf).

Evaluation

- 2 marks for correctly verifying the console input
- 4 marks for producing the correct console output for the given inputs
- 1 mark for the console output of your tests.
- -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 (10 points):

Purpose: To practice the use of conditionals and simple loops.

Degree of Difficulty: Moderate. The code is not very difficult, but make sure you understand the described program behavior.

The game of **Morra** is a hand game that dates back to ancient Greece and Rome. In the real game, any number of players can play, but for this question we will assume two players only. The game is played in rounds. Each round, both players throw out one hand showing between 0 and 5 fingers and simultaneously call out what they think the **sum** of the fingers on the hands of **all** players will be. Any player who guesses correctly gets a point. The first player to 3 points wins.

You will write a program that simulates a two-player game of Morra. Your program must do the following for each round of play until the game is over:

- At the beginning of each round, print out the round number. The first round is round 1.
- Read from the console the number of fingers shown by player 1, the number of fingers shown by player 2, player 1's guess at the sum, and player 2's guess at the sum.
- Print to the console whether any players made a correct guess, and if so, that player's new point total. If neither player guessed correctly, print a message indicating this. Note that BOTH players might guess correctly, in which case they both earn a point!

Once the game is over, print the final outcome of the game. There are a few possibilities:

- Print `Player X wins!` where X is either 1 or 2 depending on which player won.
- If, however, the winning player won by a score of 3 to 0, instead print `Player X wins a glorious victory!`, again where X is either 1 or 2 as appropriate.
- It is possible that the game is a tie. For example, if the score is 2 to 2, and both players guess correctly in the next round, both players will have three points when the game ends. In such a case, instead of printing either of the above messages, print `It's a tie!`.

Sample Runs

A couple of sample runs demonstrating program behaviour are given in the appendix at the end of this document (this is not because they are unimportant, but because they are rather long to insert here).

Tips and Hints

(a) Remember: you don't have to generate player moves randomly. You are reading them from the console each round. Think of your program as the referee — it asks for the players moves each round using console input, then reports on the outcome of each round using console output, and finally prints the outcome of the game.

(b) Your program only has to play one full game. To referee another game, run the program again!

(c) You may assume that the user enters only valid data. That is, you do not have to actually check whether player moves are between 0 and 5 and that their guesses are between 0 and 10. Just assume that valid values are always entered.

What to Hand In

- A file called `a4q2.py` containing your finished program, as described above.

Evaluation

- 1 mark for printing the round number each round
- 2 marks for console input of player moves
- 4 marks for correctly printing the results of the round for all possible outcomes
- 1 mark for correctly detecting the end of the game
- 2 marks for correctly printing the final results of the game
- -1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file

Appendix: Sample Run for Morra

In our first example, player 1 has uncanny knowledge of the future and guesses correctly every time, winning a glorious victory! Note: green text was console input entered by a user.

```
Round 1:
How many fingers will player 1 hold out? 3
How many fingers will player 2 hold out? 1
What is player 1's guess of the total number of fingers? 4
What is player 2's guess of the total number of fingers? 8
Player 1 guessed correctly!  Player 1 now has 1 points.
Round 2:
How many fingers will player 1 hold out? 0
How many fingers will player 2 hold out? 5
What is player 1's guess of the total number of fingers? 5
What is player 2's guess of the total number of fingers? 3
Player 1 guessed correctly!  Player 1 now has 2 points.
Round 3:
How many fingers will player 1 hold out? 3
How many fingers will player 2 hold out? 3
What is player 1's guess of the total number of fingers? 6
What is player 2's guess of the total number of fingers? 5
Player 1 guessed correctly!  Player 1 now has 3 points.
Player 1 wins a glorious victory!
```

In our second game, both players play well, and player 1 wins 3 to 2. In round 1, both players guess wrong. Again, green text was console input entered by a user.

```
Round 1:
How many fingers will player 1 hold out? 2
How many fingers will player 2 hold out? 3
What is player 1's guess of the total number of fingers? 4
What is player 2's guess of the total number of fingers? 6
Neither player made a correct guess.
Round 2:
How many fingers will player 1 hold out? 3
How many fingers will player 2 hold out? 1
What is player 1's guess of the total number of fingers? 4
What is player 2's guess of the total number of fingers? 8
Player 1 guessed correctly!  Player 1 now has 1 points.
Round 3:
How many fingers will player 1 hold out? 5
How many fingers will player 2 hold out? 2
What is player 1's guess of the total number of fingers? 6
What is player 2's guess of the total number of fingers? 7
Player 2 guessed correctly!  Player 2 now has 1 points
Round 4:
How many fingers will player 1 hold out? 1
How many fingers will player 2 hold out? 1
What is player 1's guess of the total number of fingers? 2
What is player 2's guess of the total number of fingers? 9
Player 1 guessed correctly!  Player 1 now has 2 points.
Round 5:
```

```
How many fingers will player 1 hold out? 2
How many fingers will player 2 hold out? 4
What is player 1's guess of the total number of fingers? 3
What is player 2's guess of the total number of fingers? 6
Player 2 guessed correctly!  Player 2 now has 2 points
Round 6:
How many fingers will player 1 hold out? 3
How many fingers will player 2 hold out? 2
What is player 1's guess of the total number of fingers? 5
What is player 2's guess of the total number of fingers? 7
Player 1 guessed correctly!  Player 1 now has 3 points.
Player 1 wins!
```

Question 3 (9 points):

Purpose: To practice using nested loops to solve a problem

Degree of Difficulty: Tricky

Pikachu is surrounded by pesky zubats! Will Pikachu be able to defeat all the bats, or will their numbers prove too great for our hero? You must write a program to answer this question.

When your program starts, allow the user to input the number of zubats that are attacking Pikachu (you may assume the user enters a positive number).

Then, you must **simulate** a series of battles between Pikachu and a zubat. Initially, Pikachu has **35 health**, but he may lose health during the battles, and if he ever runs out of health, he faints!

The zubats will attack **one at a time**, and the rules for the battle simulation are as follows. Note that you will definitely need to import Python's `random` module, as there is luck in the simulation.

- The zubat attacks first. It has a 50% chance to hit Pikachu (**Hint:** Use the `random()` function to generate a random float between 0 and 1, then check if that float is less than 0.5). If the zubat hits, it will randomly deal either 1, 2, or 3 damage to Pikachu.
- Then Pikachu attacks. Pikachu has a 60% chance to hit the zubat. If he does, he will defeat the bat.

The bats will attack in turn until either they are all defeated, or Pikachu faints. When that happens, print out a message that describes the outcome.

- If Pikachu defeated all the zubats, print out that he triumped and how much health he has left
- If Pikachu fainted, print out how many of the zubats he managed to defeat.

Sample Run

Sample input and output for Pikachu winning (input typed by the user is shown in green text):

```
How many zubats are attacking Pikachu?  10
Pikachu defeated all 10 zubats!
He has 29 health left!
```

And another sample where Pikachu faints

```
How many zubats are attacking Pikachu?  20
After bravely defeating 15 zubats, Pikachu fainted.
```

Program Design

To get started, you should divide your work into two parts.

Step 1

First, write a **function** to simulate a **single battle** between Pikachu and one zubat. Pikachu's current health should be a **parameter** to this function, and the function should return the amount of health Pikachu has **remaining** after the **battle is over**.

Make sure this function is working before going on! Put print() statements into your function to print out intermediate values during the battle to see if they make sense. You'll take these print() statements out later (the sample output above doesn't have them), but they're invaluable for testing your function.

Step 2

Second, in the "main" part of your Python program, use console input to ask the user for the number of zubats. You can assume the user will enter a positive integer.

Then, use a loop to keep calling your function from Step 1 so long as there are still zubats to defeat and Pikachu hasn't fainted. Once the loop is done, display the outcome to the console as per the sample output shown above.

What to Hand In

- A file called `a4q3.py` containing your finished program, as described above.

Evaluation

- 1 mark: Your battle function has the right parameter and return value
- 3 marks: Your battle function correctly implements the rules of the battle
- 1 mark: The battle function is called the correct number of times
- 1 mark: Pikachu's health is correctly updated using the function's return value
- 2 marks: The possible outcomes are correctly reported
- 1 mark: Your code is well-documented, including a docstring for your battle function
- -1 mark if the student did not include their name, NSID, student number and instructor's name at the top of the submitted file.