CSc 110 Assignment 5: Condition-Driven Loops

Reminder: Your code is to be designed and written by only you and not to be shared with anyone else. See the Course Outline for details explaining the policies on Academic Integrity. Submissions that violate the Academic Integrity policy will be forwarded directly to the Computer Science Academic Integrity Committee.

All materials provided to you for this work are copyrighted, these and all solutions you create for this work cannot be shared in any form (digital, printed or otherwise). Any violations of this will be investigated and reported to Academic Integrity.

Learning Outcomes:

When you have completed this assignment, you should understand:

- How to define and call functions that use condition-driven loops.

Getting started

1. **DOWNLOAD** the starter fill called `assignment5.py` and open it in your Wing editor.
   This file contains constant definitions and a function definition that uses these constants. You must make use of this provided function in your solution to Problem 4 as described in the specification.
2. Design the functions according to the specifications in the Function Specifications section.

Submission

1. Double check your file before submission to avoid a **zero grade** for your submission for issues described in the Grading section of this document:
   1. Open and **run `assignment5.py`** in your Wing editor using the green arrow.
      You should see no errors and no output after the shell prompt.
      If there are errors, you must fix them before submitting.
      If there is output printed to the shell, find and remove any top-level print statements and/or top level function calls.
   2. At the shell prompt in Wing (>>>) type the following: **`import assignment5`**
      You should see no errors and no output after the shell prompt.
      If there are errors, you must fix them before submitting.
      If there is output printed to the shell, find and remove any top-level print statements and/or top level function calls.
   3. At the shell prompt in Wing (>>>), make calls to the required functions to ensure you have named them correctly. Ensure the function is exhibiting the expected behaviour when called and does not contain any unexpected output. You should be able to make calls to the following functions with expected input. Note: When making these calls, you must precede it with the module name ie. `assignment5.is_harshad_number(14)`
      i. get_sum_of_digits
      ii. is_harshad_number
      iii. get_first_n_harshad_numbers
      iv. play
2. Upload your **`assignment5.py`** containing the completed function designs to BrightSpace

Grading:

- Late submissions will be given a **zero grade**.
- The files you submit must be named **`assignment5.py`**
  The filenames must be EXACT for them to work with our grading scripts. Errors in the filenames will result in a **zero grade**.
  Example mistakes often made by students include but are not limited to: spelling errors, different case letters, space characters and incorrect extension (not using .py)
- Your function names and the order of your function arguments must match EXACTLY as specified in this document or you will be given a **zero grade**.
- Your submission must not contain any print statements that are not required in the specification or any top-level calls to functions. This unexpected code can cause the automated tester to crash and will result in a **zero grade**.
- We will do **spot-check grading** in this course. That is, all submissions are graded BUT only a subset of your code might be graded. You will not know which portions of the code will be graded, so all of your code must be complete and adhere to specifications to receive marks.
- Your code must run without errors with Python 3.9. If you tested your configuration with setup.py file this would have verified you are using Python 3.9. Code that generates errors cannot be tested and will be given a **zero grade**.

Marks will be awarded for correctness, considering:

- the function signature matches the description given (has the name and arguments EXACTLY as specified)
- the function has the expected behaviour

and for code quality according to software engineering properties such as:

- documentation in docstring format: type hints, purpose, examples
- Test coverage – examples within the function docstring cover all boundary cases of all conditions within your function
- readability
  - use of whitespace
  - splitting complex computation or long statements across multiple lines
- meaningful variable names
  - lower case, starting with an alpha-character
- proper use of constants (avoid magic numbers)
  - defined above all function definitions
  - in UPPERCASE
- use of code constructs (functions, variables, conditions) to:
  - eliminate redundant code and redundant computation
  - make complex code easier to read

Function Specifications

Examples provided are for explanation purposes. You can/should create your own example data for additional testcases to demonstrate full test coverage.

NOTE: A **zero grade** will be given for solutions that use a for-loop instead of a while-loop.

You are free to design and use additional helper functions – ensure you have provided documentation.

1. Design a function called `get_sum_of_digits` that takes an integer and returns the sum of each digit in the integer.

   If the integer is a negative value, the function should ignore the negative sign (TIP: look up the built-in `abs` function documentation ie. `help(abs)`)

   You must use math operations (integer division(//) and modulo (%) to solve this problem).

   Examples:
   `get_sum_of_digits(0)` should return the value 0

   `get_sum_of_digits(432)` should return the value 9 since 4 + 3 + 2 = 9

   432 % 10 = **2** and 432 // 10 = 43
   43 % 10 = **3** and 43 // 10 = 4
   4 % 10 = **4** and 4 // 10 = 0

   `get_sum_of_digits(-571)` should return the value 13 since 5 + 7 + 1 = 13

   NOTE: A **zero grade** will be given for solutions that convert the number to a string and access each digit element of the string.

2. Design a function called `is_harshad_number` that takes an integer value assumed to be greater than 0 and determines whether it is a *harshad* number or not. Be sure to choose the **best** return type.

   "a harshad number (or Niven number) is an integer that is divisible by the sum of its digits"
   (taken from https://en.wikipedia.org/wiki/Harshad_number)

   Examples:

   432 is a harshad number since 4 + 3 + 2 = 9 and 432 is divisible by 9
   433 is not a harshad number since 4 + 3 + 3 = 10 and 433 is not divisible by 10

   Reminder: always make use of helper functions where appropriate.

3. Design a function called `get_first_n_harshad_numbers` that takes a non-negative integer n. The function should return a string containing the first n *harshad* numbers. There must be a comma between each pair of values in the string with **no trailing comma** or your function will fail our automated tests.
   Think simple… repeatedly check one number at a time to see if it is a *harshad* number, if it is, append it to the sequence, if it is not, do not append it to the sequence. Repeat this until you have n numbers appended to the sequence.

   The call `get_first_n_harshad_numbers(0)` should return the string:
   `''`
   The call `get_first_n_harshad_numbers(1)` should return the string:
   `'1'`
   The call `get_first_n_harshad_numbers(20)` should return the string:

   `'1,2,3,4,5,6,7,8,9,10,12,18,20,21,24,27,30,36,40,42'`

4. Design a function called `play` that simulates the playing of one round of a two dice game according to the rules given below. The function should take as arguments: two integers representing the two numbers the player guesses will be rolled and an integer representing the number of dollars the player would like to bet. Your function should assume the guesses are numbers from 1 to 6 inclusive and the bet is whole number bigger than 0. The function should return the amount of money the player has after the round is over.

   Your function **must** call the `roll_one_die` function provided that returns a random number (from 1 to 6 inclusive) to represent a single dice roll. Our automated tester will depend on you calling this function and if you do not **a zero grade** will be given. See the comments in `roll_one_die` function help with testing versus running a realistic version of the function.

   You should omit examples from your documentation due to the behaviour being dependent on randomly generated values with the calls to `roll_one_die` function.

   DO NOT prompt the user for input! The function is to take inputs as arguments. Prompting the user for input in any way will cause our tester to fail and result in a zero grade for this function.

   Rules:
   - A pair of dice are rolled for the player
   - If both dice rolled matches the player's guesses, the round is won and the player triples their money.
   - If neither dice rolled matches the player's guesses, the round is lost and the player loses the money they bet.
   - If one of the dice rolled matches one of the player's guesses, then the sum of the 2 dice values is calculated, this is called the losing target. Only in this case the game continues and the pair of dice continues to be rolled until either:
     - The sum of the dice rolled matches the losing target, even if one of the dice matches the guess, the round is lost.
     - One of the dice rolled matches the player's other guess and the sum of the two dice rolled does not match the losing target, in this case the round is won.

The following pages provide sample runs of calling `play` on a model solution.

Notice, each sample run shows the function call and sample screen output and then tells you what the expected returned value would be.

The wording of the screen output in your program does not need to match ours exactly, you can and should customize your output. It is the value returned from the function that our automated tester will inspecting, not the wording of your printed output. BUT, you must **NOT prompt the user** for input anywhere – doing so will result in our tester timing out and you will receive a zero grade.

NOTE: These sample runs are provided for clarification purposes. Your function must work for all other expected argument values and possible dice rolls.

Sample Runs

Sample Run A:

```
play(3, 2, 40)

you guessed 3 and 2 will be rolled and bet $40!

you rolled 3, 2

You guessed both correct!
You now have $120!
```

The function should return 120 in this case, since $40 was bet and they tripled their money.

Sample Run B:

```
play(3, 2, 50)

you guessed 3 and 2 will be rolled and bet $50!

you rolled 2, 3

You guessed both correct!
You now have $150!
```

The function should return 150 in this case, since $50 was bet and they tripled their money.

Sample Run C:

```
play(3, 2, 40)
you guessed 3 and 2 will be rolled and bet $40!

you rolled 3, 6

You guessed one right!
You get a second chance to roll the dice for your other guess: 2
Don't let your roll total 9 or your second chance is over and you lose

you rolled 6, 2

Your second chance is over...
But luckily you rolled a 2 during your second chance!
You now have $80!
```

The function should return 80 in this case, since $40 was bet and they double their money.

Sample Run D:

```
play(3, 2, 100)
you guessed 3 and 2 will be rolled and bet $100!

you rolled 2, 5

You guessed one right!
You get a second chance to roll the dice for your guess: 3
Don't let your roll total 7 or your second chance is over and you lose

you rolled 4, 2
you rolled 5, 6
you rolled 3, 5

Your second chance is over...
But luckily you rolled a 3 during your second chance!
You now have $200!
```

The function should return 200 in this case, since $100 was bet and they double their money.

Sample Run E:

```
play(3, 2, 100)
you guessed 3 and 2 will be rolled and bet $100!

you rolled 2, 5

You guessed one right!
You get a second chance to roll the dice for your guess: 3
Don't let your roll total 7 or your second chance is over and you lose

you rolled 1, 6
oh no - you rolled the losing target!

Your second chance is over...
So sorry you lost - come again soon!
You now have $0!
```

The function should return 0 in this case, since $100 was bet but they lost.

Sample Run F:

```
play(3, 2, 100)
you guessed 3 and 2 will be rolled and bet $100!

you rolled 3, 4

You guessed one right!
You get a second chance to roll the dice for your guess: 2
Don't let your roll total 7 or your second chance is over and you lose

you rolled 2, 5
oh no - you rolled the losing target!

Your second chance is over...
So sorry you lost - come again soon!
You now have $0!
```

The function should return 0 in this case, since $100 was bet but they lost.