COMP 1900 - Fall 2021

Lab 5: Loops Episode II

Total Points: 36

Due: Fri., Oct. 29, by 2359 CDT
Please carefully read the submission instructions at the end of the assignment. As with all other assignments, this should be done *individually*.

Grader: The lab TA who supervises your registered lab section will grade your submission. Grades will generally be posted within 1-2 weeks of the assignment due date. Questions about grading? Please contact them first.

1. (4 points) Write a program named `dice_rolling.py` that analyzes rolling two dice. The program should start by getting user input for the number of sides on each die. Each die is numbered from 1 up to its number of sides, inclusive. Note that the two dice *can* have different numbers of sides. You may assume that the user will enter a valid number of sides.

   After the user specifies the number of sides, the program should list all possible outcomes of rolling those two dice. The order of outcomes on the individual dice *does* make a difference; for example, rolling 1 on the first die and 2 on the second is a different outcome from rolling 2 on the first and 1 on the second. Also print how many total outcomes are possible.

   Note that this program is not simulating the dice roll, so no random number generation is needed.

   Example program run (underlined parts indicate what the user enters)

```
Enter number of sides on first die: 2
Enter number of sides on second die: 3

There are 6 possible outcomes from rolling 1d2 and 1d3:
(1, 1)
(1, 2)
(1, 3)
(2, 1)
(2, 2)
(2, 3)
```

2. (8 points) Modify your `guessing_game.py` program from the previous lab as follows:

   - After the player has finished the game, ask if they'd like to play again. If so, restart the game with a new random number, *without* changing the range of possible numbers that was originally entered by the player.
   - After the player finishes each game, show statistics on the lowest, highest, and average number of guesses taken by the player over all games played.

   The other features (how the game works, input validation, etc.) should be unchanged.

   Example program run (underlined parts indicate what the user enters)

```
Let's set up our guessing game!
I'll pick a random integer between a certain min and max (inclusive)
 that you specify.

Min: 2
Max: 4

OK, let's play!
I'm thinking of an integer between 2 and 4, inclusive.
Enter your guess: 2
Woohoo, you got it!  It only took you 1 try.
You should consider a career in the psychic industry.

Your stats so far:
Minimum guesses taken: 1
Maximum guesses taken: 1
Average guesses taken: 1.0

Play again? (Y for yes, anything else for no): Y

OK, let's play!
I'm thinking of an integer between 2 and 4, inclusive.
Enter your guess: 4
Too high!  Try again...
Enter your guess: 3
Woohoo, you got it!  It only took you 2 tries.

Your stats so far:
Minimum guesses taken: 1
Maximum guesses taken: 2
Average guesses taken: 1.5

Play again? (Y for yes, anything else for no): N
```

3. (12 points) Credit card numbers are not completely random sequences; they follow certain rules depending on the card issuer. An American Express (AmEx) number must meet these criteria:

   - Begin with 34 or 37
   - 15 digits in length
   - Satisfy the **Luhn formula**, created by IBM scientist Hans Peter Luhn in the 1950s

   Here's how the Luhn formula works:

   - Starting from the next-to-last digit, double every other digit going left.
   - For each of the doubled values that exceed 9, subtract 9.
   - Add up all the doubled values, along with the digits that were not doubled.
   - If the result is a multiple of 10, the number satisfies the Luhn formula. If the result is not a multiple of 10, the number does not satisfy the Luhn formula.

   For example, 372366881355205 is a *valid* AmEx number. (Don't worry, this was randomly generated and probably doesn't actually belong to anyone :) We can easily verify that the number begins with 37 and is 15 digits in length. As for the Luhn formula, we start by doubling every other digit going left, starting from the next-to-last digit:

   $$3\ \ \underline{14}\ \ 2\ \ \underline{6}\ \ 6\ \ \underline{12}\ \ 8\ \ \underline{16}\ \ 1\ \ \underline{6}\ \ 5\ \ \underline{10}\ \ 2\ \ \underline{0}\ \ 5$$

   For each of the doubled values that exceed 9, subtract 9:

   $$3\ \ \underline{5}\ \ 2\ \ 6\ \ 6\ \ \underline{3}\ \ 8\ \ \underline{7}\ \ 1\ \ 6\ \ 5\ \ \underline{1}\ \ 2\ \ 0\ \ 5$$

   Adding up all the modified digits results in $3+5+2+6+6+3+8+7+1+6+5+1+2+0+5 = 60$, which is a multiple of 10. Therefore, the number satisfies the Luhn formula as well.

   Write a program named `amex_validator.py` that gets user input for a credit card number and determines whether or not it's a valid AmEx number. Read the input as a string, and use a loop to go through the digits in the user input.

   The program must show whether or not each of the three criteria is met. Each criterion should be handled *independently* – for example, even if the number doesn't begin with the correct digits or have the correct length, it should still be checked to see if it satisfies the Luhn formula. Your program should also verify that the user's input doesn't contain any invalid (i.e., non-digit) characters. If it does, it fails the "15 digits in length" criterion, the Luhn formula criterion, and possibly the "first two digits" criterion as well.

   Hints:

   - Given a string `s`, `s[i]` gives you the character at index `i` of `s`. You can use this to isolate individual digits from the user input. These digits are treated as strings, so be sure to convert them to integers using `int(...)` before doing calculations on them.
   - Given a string `s`, `s.isdigit()` results in a value of `True` or `False` depending on whether `s` is a digit.

Example program run (underlined parts indicate what the user enters)

```
Enter a card number: 372366881355205
That's a VALID AmEx number!

Begins with 34 or 37    - YES
15 digits in length     - YES
Satisfies Luhn formula  - YES
```

Example program run (underlined parts indicate what the user enters)

```
Enter a card number: 4024007169758918223
That's an INVALID AmEx number!

Begins with 34 or 37    - NO
15 digits in length     - NO
Satisfies Luhn formula  - YES
```

Example program run (underlined parts indicate what the user enters)

```
Enter a card number: 371234567890123
That's an INVALID AmEx number!

Begins with 34 or 37    - YES
15 digits in length     - YES
Satisfies Luhn formula  - NO
```

Example program run (underlined parts indicate what the user enters)

```
Enter a card number: 34abcdefghijklm
That's an INVALID AmEx number!

Begins with 34 or 37    - YES
15 digits in length     - NO
Satisfies Luhn formula  - NO
```

Example program run (underlined parts indicate what the user enters)

```
Enter a card number: 78q34ujasi;erjask;lj4sa4u
That's an INVALID AmEx number!

Begins with 34 or 37    - NO
15 digits in length     - NO
Satisfies Luhn formula  - NO
```

4. (12 points) Now that we know how to validate AmEx numbers, it's not hard to *generate* valid numbers. Here's an algorithm to do so:

   - Randomly choose 34 or 37 as the first two digits.
   - Randomly generate the next 12 digits (i.e., all the digits of the card number except the last one); each digit is between 0-9.
   - Pick the last digit to make the number satisfy the Luhn formula.

   For example, suppose we pick 34 as the first two digits, followed by 123456789012 as the next 12. Call the last digit x. To pick x such that the number satisfies the Luhn formula, start by looking at the number so far:

   `3 4 1 2 3 4 5 6 7 8 9 0 1 2 x`

   Double every other digit going left, starting from the next-to-last digit:

   $\texttt{3}\ \ \underline{\texttt{8}}\ \ \texttt{1}\ \ \underline{\texttt{4}}\ \ \texttt{3}\ \ \underline{\texttt{8}}\ \ \texttt{5}\ \ \underline{\texttt{12}}\ \ \texttt{7}\ \ \underline{\texttt{16}}\ \ \texttt{9}\ \ \underline{\texttt{0}}\ \ \texttt{1}\ \ \underline{\texttt{4}}\ \ \texttt{x}$

   For every doubled value that exceeds 9, subtract 9:

   $\texttt{3}\ \ \texttt{8}\ \ \texttt{1}\ \ \texttt{4}\ \ \texttt{3}\ \ \texttt{8}\ \ \texttt{5}\ \ \underline{\texttt{3}}\ \ \texttt{7}\ \ \underline{\texttt{7}}\ \ \texttt{9}\ \ \texttt{0}\ \ \texttt{1}\ \ \texttt{4}\ \ \texttt{x}$

   Adding up all the modified digits gives $63 + x$. Remember that we want the final sum to be a multiple of 10, so we should pick a value of x between 0-9 such that $63 + x$ is divisible by 10. Clearly, $x = 7$ satisfies this. To get the final valid AmEx number, just go back to the original set of digits (34123456789012) and append 7 to the end: 341234567890127.

   Write a program named `amex_generator.py` that asks the user for how many AmEx numbers they want, and then uses the generation algorithm from this problem to produce and show the numbers on the screen. Include input validation with a loop to ensure that the user must request at least one number. To verify that your generated numbers are valid, you can use your code from the previous problem, and/or check them at `https://www.freeformatter.com/credit-card-number-generator-validator.html`.

   This problem probably seems awfully shady, but keep in mind that having valid card numbers doesn't mean much by itself. First, there are 2×10^{12} (2 *trillion*) possible valid AmEx numbers, which is far beyond the number of active AmEx accounts in the world. So it's unlikely that a randomly generated number actually belongs to anyone. Second, on the off chance that you produce someone's real card number, it's generally not usable without additional personal information and the security code printed on the physical card.

   Hint: Focus on generating *one* valid number first. Once that works, just put your generation code into a loop that runs a specific number of iterations. It's theoretically possible for this to produce duplicate numbers, but it's exceedingly unlikely so don't worry about that scenario.

Example program run (underlined parts indicate what the user enters)

```
How many AmEx numbers would you like today? -3
Must enter at least 1!  Try again: 10

Here you go, have fun:
370058096443106
377183306261677
345237253774386
343626202314629
347227408751353
347316687984486
348659155142551
378183345551566
372053455450139
342416243448094
```

Code Guidelines

Points can be deducted for not following these guidelines!

- Most importantly, your code *must* run. Code that does not run due to syntax errors may receive zero credit, at the TA's discretion.
- Follow Python capitalization conventions for `variable_and_function_names`.
- Use consistent indentation throughout your code. (Python kind of forces you to do this!)
- Include a reasonable amount of comments in your code. "Reasonable" is somewhat subjective, but at the very least include:
  1. A comment at the top of each program summarizing what it does.
  2. Comments that indicate the major steps taken by the program. There are generally at least two or three of these, such as collecting user input or making calculations.

Need Help?

- Attend your weekly lab meeting. Your lab TA can give you live help during this session.
- Contact your lecture instructor, and/or attend office hours.
- The UofM offers free online tutoring through the Educational Support Program (ESP): `https://www.memphis.edu/esp/onlinetutoring.php`

Submission Instructions

- Create a zip file containing your Python source files. You can use any Python development environment you like, as long as you submit the source files.
- Upload your zip file to the appropriate dropbox folder on eCourseware. The dropbox *will* cut off submissions at precisely the stated deadline, so please submit with some time to spare. Late submissions are not accepted. You may submit as many times as you want up to the deadline. Each submission overwrites the previous one.
- Contact your lab TA to schedule a one-on-one code review via Zoom. During this short meeting (15-30 minutes), your TA will ask you to run your code. They may also ask you to explain your code and/or run different test cases. The code review must be completed for you to get a grade for the lab.

  Note that since the code review happens after your submission, its purpose is *not* for you to get help with writing your code! Getting help should be done during the scheduled lab sessions, or by reaching out to your lecture instructor/online tutors *before* the due date.