Math 260, Fall 2020
Last updated: 8/21/20

HOMEWORK 1

DUE FRI. AUG 28

Submission: Submit your code (as noted in the problems) to gradescope.

Notes: Exercises are short pieces of code. Ingeneral, they will be submitted togethe in a **single python file.** Here, there is nothing to submit for the exercises!

You are given some template code (see the `hw1` folder in the course github).

Exercises.

I1 (input/output, no submission). Data for the batting averages of a team of blaseball players is supplied in the `build_player_data` function.

Write a function `print_avg())` that displays this data in the form of a **three-column** table that looks something like this (you can adjust the exact look):

```
player   num    avg.
name1    13    .356
name2     5    .123
```

The average should use three digits; assume player numbers are always 1 or 2 digits. Note that the table should be easy to read, but you don't have to make it line up exactly.

Do this two ways: (i) using an f-string and (ii) using `.format`. In the future, use whichever method you prefer.

As a hint, you can often use tabs ('\t') to force entries to align, e.g.

```
print('aaa \t 33')
print('b \t 5')
```

Programming problems (submit code).

P1 (a fibonacci like example). First, the context...

In a game of Texas Hold 'Em, a player is dealt two cards as their starting hand. The probability of being dealt a pair of aces is

$$a = \frac{4}{52} \cdot \frac{3}{51} \approx 0.0045249. \tag{1}$$

It's even more unlikely that one will be dealt this hand twice in a row. However, over the course of many hands, the chances of having such good luck happen will go up (runs of unlikely events become likey with enough trials!). Let

$$p_n = \text{probability of seeing a pair of aces two hands in a row in } n \text{ hands.}$$

and let $q_n = 1 - p_n$.

It's a straightforward probability exercise to show that[1]

$$q_n = (1-a)q_{n-1} + a(1-a)q_{n-2}, \qquad q_0 = q_1 = 1. \tag{2}$$

... and now, the computational work...

a) Write a function `prob_list(n, a)` that calculates q_n using (2), building a list of values $q_0, \cdots, q_n$ and then returning the last one.

b) How many hands must be played to have at least a 50% chance of two pairs of aces in a row? Write code to compute this and a main function that executes the code (so running your code outputs the answer to this part).

c) Now write another version, `prob(n, a)` that calculates q_n without creating a list. Instead, it should only track the previous two values (you'll need three variables for this).

Hint: You'll need variables `a`, `b` and `c` (rename as you see fit) that represent the 'previous-previous' (q_{n-2}), the 'previous' (q_{n-1}) and the 'current (q_n) values. At each iteration of your loop, all three of these variables must be updated.

Remark: A crude estimate can be made by saying that this event occurs with probability a^2 in two hands. There are roughly n sets of two consecutive hands, hence the expected number of pairs-in-a-row is 1 if $n = 1/a^2$. This gives you a sense of the 'order of magnitude' for (c).

[1] To do so, condition on the last 'failure' (not having a pair of aces) in the n hands.

P2 (Rock-paper-scissors). Your task here is to create a game of rock-paper-scissors that a user can run and play (against the computer).

a) (**The main loop**) Write a function `rps()` with no return that does the following:

- The user plays **rounds** of the game. In each **round**,

  - the user's move is queried using `input` - either rock, paper or scissors until a valid move is entered. (Input format: your choice; keep it simple).

  - the computer chooses a move (see part (b))

  - the moves and the winner is displayed

- A list of the moves up to the current round is maintained. Use a list of lists, e.g.

```
[[0, 1], [1, 2], [1, 0]]
```

  Internally (in the code), use 0, 1 and 2 to represent rock/paper/scissors (this will be easier to work with than strings).

- After each round, the player is asked if they want to continue (y/n).

- At the end, a summary of the game is displayed (see part (c)).

b) (**The computer's choice**) For the computer's move, write a function `cpu_move(moves)` that tkaes in the moves list and returns the computer's move. The strategy is up to you; not that you don't have to use all of the previous moves data for the computer to decide.

For bonus credit, the computer should always have an advantage if the user always guesses the same thing (only rock, only paper or only scissors).

c) (**The summary**) After the player stops, the program should call a function `summary` that takes in the data from the game and displays (i) the moves made by the player and computer and who won (in three columns) and (ii) the overall winner and their win %.

The output would look something like:

```
player  cpu
-------------------
paper    rock    1
paper    rock    1
rock     rock    T
scissors rock    0
winner: player!
win pct: 67%
```

Remark: There's a number of ways you can organize to deal with the annoyance of the case-work. One way is to make a table of outcomes (a matrix whose (i, j) entry tells you who wins given move i vs. move j). There are other ways to do so as well, which are up to you (and brute force case work is fine. as long as the code is not redundant).