CSC148, Assignment #1
due January 30th, 2017, 10 p.m.

overview

Assignment 1 begins a series of two related assignments on computerized games. In this assignment you will hone your skills at designing and implementing classes, including inheritance. You will end up with a simple game interface that will pit you against a weak computer opponent.

Assignment 2 will continue this theme. You will implement classes that provide a stronger computer opponent, face efficiency challenges, and add more complex games.

games background

In these assignments you will be programming computers to play games of a restricted type, namely two-player, sequential-move, zero-sum, perfect-information games. Lots of games have these features: tic-tac-toe, chess, go, checkers, mancala, and nim, for example. Important characteristics are:

two-player: There are two distinct players. We'll call them Player A and Player B.

sequential-move: Players take turns making moves, one after the other, *i.e.*, in sequence. Moves change the state of the game. (For example, in tic-tac-toe, the state of the game is the configuration of Xs and Os, together with which player is about to take a turn.) The only possible outcomes of the game are a win for Player A (which is a loss for Player B), a win for Player B (which is a loss for Player A), or a tie. In some games, no tie is possible.

zero-sum: The benefit of any move for Player A is exactly inverse to that move's benefit for Player B. For example, a move that wins for Player A loses for Player B. A move that takes Player A closer to a win takes Player B closer to a loss. If we can measure and add the benefit for Player A to the benefit for Player B of any move, the result is zero.

perfect-information: No information is hidden. Both players know everything about the game state, and all moves made by their opponent. Compare this to a typical card game, where players keep the cards in their hand hidden from the other players.

description of classes

the state of the game

You'll need code to keep track of the game state, or a snapshot of the current situation in the game. The game state will need to represent which player is currently playing, which legal moves (if any) are available to that player, and whether the game is over. For example: In checkers, you would want to keep track of whether White or Black is playing, and which pieces are where. From this information, you could determine all available moves for the current player. That is, the "state" determines the game. If a player has no pieces left in checkers, you can determine that the game is over, and the player who still has pieces is the winner. It must be possible to go from one game state to another by making a legal move, if there is one.

Design your code so that it is not specific to a particular game. We want it to be general so that in a later assignment you will be able to add additional games and to let the user choose which one to play. But here's a paradox: a generic game state cannot possibly know things such as what are the legal moves available, whether the game is over and who has won, because it doesn't know which particular game is being played. You will deal with the paradox using inheritance: implement the game state features that are common to all games in a generic game state class, but implement game-specific features in a subclass for each game. As you will see in lecture, Python provides a **NotImplementedError** exception that you may use in the generic class for features that you know are necessary, but that can only be implemented in subclasses.

The exact design and naming of your classes is left up to you. The constraints are that they must allow the code we provide you in **game_interface.py** to play **subtract square** and **chopsticks**, both described below.

the computer's strategy

You'll also need to complete code, in **strategy.py**, that uses some strategy to choose a move for the computer to make. For Assignment 1, this will be an extremely simple strategy, randomly choosing one of the legal moves available. Your strategies must work for as-yet-unspecified games, so they should have no game-specific detail. In the final version of your code, for Assignment 2, you will let the user choose which strategy the computer will use.

the overall play of the game

Finally, we provide you with code in **game_interface.py** that manages the player's "view" of the game. It allows a user to play a complete game against a computer opponent. The player's view will be text-based. It will include informing them of the aim of the game and relevant rules, prompting for a move, indicating whether a chosen move is legal, showing what move the computer has chosen, and indicating whether a player has won.

By carefully reading our code in **game_interface.py** you will be able to figure out what the classes that you design must provide.

Be sure to modify **only** the portions we indicate in **game_interface.py** and **strategy.py**, or else we will not be able to test your code.

subtract square

You'll implement **subtract square**, a game which is played as follows:

1. A non-negative whole number is chosen as the starting value by some neutral entity. In our case, a player will choose it (i.e. through the use of input()).

2. The player whose turn it is chooses some square of a positive whole number (such as 1, 4, 9, 16, ...) to subtract from the value, provided the chosen square is not larger. After subtracting, we have a new value and the next player chooses a square to subtract from it.

3. Play continues to alternate between the two players until no moves are possible. Whoever is about to play at that point loses!

chopsticks

You'll implement **chopstick**, a game played as follows:

1. Each of two players begins with one finger pointed up on each of their hands.
2. Player A touches one hand to one of Player B's hands, increasing the number of fingers pointing up on Player B's hand by the number on Player A's hand. The number pointing up on Player A's hand remains the same.
3. If Player B now has five fingers up, that hand becomes "dead" or unplayable. If the number of fingers should exceed five, subtract five from the sum.
4. Now Player B touches one hand to one of Player A's hands, and the distribution of fingers proceeds as above, including the possibility of a "dead" hand.
5. Play repeats steps 2–4 until some player has two "dead" hands, thus losing.

your job

We provide you with starter files

- game_interface.py, where you have to complete the "TODOs" and reason about what classes you need to design to make this interface work.
- a1_pyta.txt, to specify exceptions from python_ta's default settings.
- strategy.py, where you have to complete a "TODO" for a random move-choosing strategy
- chopsticks_unittest_subset.py, subtractsquare_unittest_subset.py, which should increase your confidence that basic features of your chopstick and subtract squares game work.

You will design and implement classes to support the simple game-playing interface described above. Class design, names, and other implementation choices are left up to your taste, applying the concepts you have learned in this course. However, we insist on the following:

- All your *.py files must contain, in the **if __name__** == "**__main__**": block, the lines:

  ```
  import python_ta
  python_ta.check_all(config="a1_pyta.txt")
  ```

  ... and produce no errors or warnings when your file is run.
- You must implement appropriate **__init__**, **__str__**, and **__eq__** methods for each class that you define.
- We must be able to play your games, on a teach.cs machine, by typing

  ```
  $ python3.6 game_interface.py
  ```

  The command above is appropriate on unix-like platforms. Another way of saying this is **we must be able to**:

  1. Open **game_interface.py** in Pycharm.
  2. Click the green arrow ("run")
  3. ... in order to play your game.

Submitting your work

Submit all your code on MarkUs by 10 p.m. January 30 — or earlier! You may submit **multiple** versions of your files, and only the most recent version is marked. Click on the “Submissions” tab near the top. Click “Add a New File” and either type a file name or use the “Browse” button to choose one. Then click “Submit”. You can submit a new version of a file later (before the deadline, of course); look in the “Replace” column.

Submit:

- **game_interface.py**
- **strategy.py**
- ...plus **all** files containing classes that support your implementation of these games.