CSC148, Assignment #1
due January 29th, 2015, 10 p.m.

deadline to declare your assignment team: January 22nd, 10 p.m.

overview

Assignment 1 begins a series of three 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.

Assignments 2 and 3 will continue this theme. You will implement classes that provide a stronger computer opponent, face efficiency challenges, and add more complex games.

games background

Humans and other animals engage in many activities that aren't directly essential to life (those would be work), but are pursued for the sheer joy that comes from facing, and possibly overcoming, a challenge: moving a ball across a field, moving your body and a skipping rope around each other, finding the word that fits in a crossword, or tweaking a Python program. Some of these recreational activities have the structure of a game. Computers can be programmed to play some games, with varying degrees of success. Some reasons for teaching computers to play games are:

- It's useful, for example, for training human players
- Programming a computer to play helps us understand the game better.
- Programming a computer to play helps us to understand things, other than games, that computers can potentially do.
- Programming a computer to play hard games can be challenging, interesting, and fun for its own sake. Some game-playing strategies for computers are completely "solved"[1], meaning that the computer plays the strongest possible game, others aren't.

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 (usually 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.

[1] For example checkers, which has roughly half a trillion game positions, was solved by researchers in Alberta in 2007.

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, which is a snapshot of the current situation in the game. The game state will need to represent which player moves next, which legal moves (if any) are available to that player, whether the game is over, and whether a particular player has won. Even when there are no legal moves left, by convention the next player is still defined to be the opponent of whichever player just played. In addition, 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.

As part of Assignment 1, in addition to the generic game state class, you will need to implement a game state class for the specific game **Subtract Square** (see below).

the computer's strategy

You'll also need to create code that uses some strategy to choose a move for the computer to make. For Assignment 1, this may be an extremely simple strategy, such as 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. However, you must design your code so that it will be easy to add more strategies. In the final version of your code, for Assignment 3, you will let the user choose which strategy the computer will use.

Again, use inheritance to create generic code that implements features that are common to any strategy for choosing the next move. Then you can create a subclass corresponding to each specific strategy. For Assignment 1, you are required to implement only one specific strategy.

the overall play of the game

Finally, you'll need code that manages the player's "view" of the game. It must allow 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. The details

of the player's view are up to you, but make sure that a friend who doesn't know the game would find it playable.

For Assignment 1, your game view will use the specific game **subtract square** and the one strategy that you will have implemented so far. In future assignments, it will ask the user to choose the game and the strategy.

subtract square

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

1. A positive whole number is randomly chosen as the starting value by some neutral entity. In our case, the computer will choose it randomly.
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!

your job

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:

- You must follow CSC108 style guidelines unless your CSC148 instructors say otherwise.
- You must implement appropriate **__repr__**, **__str__**, and **__eq__** methods for each class that you define.
- You must have both a class for a generic game state, and a subclass for a subtract square game state. Your generic game state should be designed so that additional subclasses for other specific, as-yet-unspecified, games can be added.
- You must have both a generic class for choosing moves, and a subclass that uses some really simple strategy to choose moves.
- You must have a class for the game view. It must be saved in a file called **game_view.py** and **we must be able to invoke**:

  ```
  $ python3 game_view.py
  ```

  ... **in order to play subtract square against a computer opponent.**

  [Edit: January 20th] The command above is appropriate on unix-like platforms. Another way of saying this is **we must be able to**:

  1. Open **game_view.py** in Wing.
  2. Click the green arrow ("evaluate")
  3. ... in order to play **subtract square** in Wing's Python shell against a computer opponent.

Declaring your assignment team

You may do this assignment alone or in a team of either 2 or 3 students. Your partner(s) may be from any section of the course on St George campus. You must declare your team (even if you are working solo) using the MarkUs online system.

Navigate to the MarkUs page for the assignment and find "Group Information". If you are working solo, say so. If you are working with other(s):

First: one of you needs to "invite" the other(s) to be partners, providing MarkUs with their cdf user name(s).

Second: the invited student(s) must accept the invitation.

Important: there must be **only** one inviter, and other group members accept **after** being invited, if you want MarkUs to set up your group properly.

To accept an invitation, find "Group Information" on the appropriate Assignment page, find the invitation listed there, and click on "Join".

Submitting your work

Submit all your code on MarkUs by 10 p.m. January 29. 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.

Only one team member should submit the assignment. Because you declared your team, all of you will get credit for the work.

Be sure to hand in all of your file `game_view.py` and all your other `.py` files. Once you have submitted, check that you have submitted the correct version; new or missing files will not be accepted after the due date.