Project 1: automating greed

Due: Tuesday, September 26, 11:55 pm.

Starting code folders as a ZIP package (version-specific):
* package for Python 3.4 (project1_34.zip)
* package for Python 3.5 (project1_35.zip)
* package for Python 3.6 (project1_36.zip)

In this project you will make a program that plays a dice game called **Greed**. We'll begin by with an interactive program that allows two players to play the game against each other. From there, we will build tools that allow us to explore different strategies for playing the game, by considering computer players written as Python code. The first tool will allow us to pit two computer players against each other. The second tool we'll use to figure out how well different computer players play the game. Finally, you'll write your own strategy code (or several strategy codes, if you like), written as a Python function, that automatically plays the game. You can pit this strategy function against some test players that we've written. You'll also have a chance to pit your strategy function against your classmates'. Have fun!

Download the starting code using the appropriate link just above. There are several versions of this package, and you should download the one that matches the version of Python 3 you've installed on your machine. It will download as a "ZIP package" and this will unpack as a folder (usually by just double-clicking it) on your computer's filesystem. Several versions are necessary because we've provided a pre-compiled collection of game strategies you can test, but we'd like their exact code to be a mystery to you. This compiled code is version-specific.

The Rules

The game of **Greed** is a dice game between two players. The game begins with both players having a score of zero, and they each take turns rolling dice and (possibly) increasing their score. The dice used are 6-sided, but they differ from regular ones: their sides are *labeled with the numbers 0 through 5, rather than from 1 to 6*. A player's goal is to get the higher score, though ties can happen.

The game begins with the first player choosing how many dice he'd like to roll in his turn. He can choose any number of dice: 0, 1, 2, or more. He then rolls that number of dice and adds their sum to his score. His turn is over. Next, player two makes her turn. She also chooses a number of dice she would like to roll, rolls them, and adds their sum to her score. The game continues this way, with the two players taking these alternating turns.

The game ends if, as a result of a dice roll, a player's score goes over 100. In that case, the other player wins.

There is another way the game ends: on his/her turn a player is allowed to choose to roll 0 dice, in which case, that player is choosing to *pass*. In the case that a player passes their turn, the other player *gets one more turn to roll* the dice and add that roll's total to their score. After that last roll, the player with the highest score not higher than 100 is declared the winner. They tie if they are at the same score.

The Assignment: a series of steps

Below I outline a series of steps I'd like you to follow in order to successfully complete this assignment. They are designed so as to introduce you to game play, and to help you devise strategies for playing the game. We are providing code that you will work with. Also, you will need to write code that works with ours, as well as others' code. The steps below are also designed to help you figure out these kinds of logistics.

Before you scan through these steps, you'll want to download the project folder which contains the code that you'll be working with. The first steps have you running this code.

step 1: experiment. You should first work to understand the game. The `play` function written in `greed.py` conducts a game between two human players, both typing their moves into prompts in the console. Start up the Python interactive interpreter, loading this Python file, and then evaluate the expression `play()`, similar to the commands I used on my computer below:

```
python3 -i greed.py
>>> play()
```

This will start the two-player game, immediately asking for the two players' names, and then letting them play the game. Play several games with a friend (or just against yourself) to get a feel for **Greed**. You should then look at the code for the `play` function and try to understand how it works. (It is purposefully not documented very well.)

step 2: inspect a sample strategy. Our goal is to program strategies so that they can play the **Greed** game against each other, rather than just having humans play the game. The source code `greed.py` contains the definition of a function named `sample1`. This is the code for a sample **Greed** strategy. This strategy function, like all the strategies we will write, takes three parameters. The parameters represent the current state of the game just before a player is about to take a turn. The strategy uses this information to determine its automated player's dice roll choice.

The first parameter `myscore` gives the strategy's current score. The second parameter `theirscore` gives the opponent's score. The third paramter `last` is a boolean indicating whether or not the opponent passed on their last turn. If `last` is `True` then the strategy's player is about to take its last turn to roll dice in the game.

The `sample1` function uses this info to return an integer value. That integer represents its next move, that is, how many dice it wishes to roll on this turn. If it returns 0, that means that it is choosing to pass on this turn.

You should look at this `sample1` code. It is, in fact, a very simple strategy---it passes if it's currently ahead, and otherwise it rolls 12 dice. Unfortunately, we can't actually watch it play a game given the code as it currently is in the file. We will do that in the next step. You'll also see the start of a definition for a second sample strategy named `sample2`. We'll ask you to write that code soon, too.

step 3. autoplayLoud. This step is the first coding you'll need to do for the assignment. Your job is to write a function called `autoplayLoud` that conducts games between two computer strategies instead of two people. It should work very similarly to `play`, but it will not prompt for any human input. It will instead use two automated players to make dice roll choices, one acting as Player 1 and the other acting as Player 2. That is, rather than pitting human players against each other with the `play` function, `autoplayLoud` lets us watch the gameplay of automated strategies we invent.

To do this, you'll write `autoplayLoud` so that it is fed two strategies as parameters---formal parameters named `strat1` and `strat2`. These will each be a strategy function, one that takes three parameters (just like how `sample1` was written) and returns a number of dice to roll. This makes `autoplayLoud` a great example of a higher-order function. We simply feed it two strategy functions as actual parameters, whatever ones we want to have comptete. Our code inside `autoplayLoud` can call them with `strat1(...)` or `strat2(...)` to see what they each want to do in their turn's situation. For example, if we evaluate

```
>>> autoplayLoud(sample1,sample1)
```

then we expect to see a transcript that results from a game of **Greed** where Player 1 follows the strategy coded in `sample1` and where Player 2 does too. You get to watch how `sample1` fares against itself.

Here is what that might look like:

```
Player 1: 0   Player 2: 0
It is Player 1's turn.
12 dice chosen.
Dice rolled: 3 4 5 2 3 2 1 0 1 5 2 4
Total for this turn: 32

Player 1: 32   Player 2: 0
It is Player 2's turn.
12 dice chosen.
Dice rolled: 4 2 4 2 4 1 3 1 4 0 1 5
Total for this turn: 31

Player 1: 32 Player 2: 31
It is Player 1's turn.
0 dice chosen.
Dice rolled:
Total for this turn: 0

Player 1: 32   Player 2: 31
It is Player 2's turn.
12 dice chosen.
Dice rolled: 0 3 4 1 1 0 4 2 5 0 2 2
Total for this turn: 24

Player 1: 32   Player 2: 55
Player 2 wins.
```

Write the code for `autoplayLoud` by adapting the code for `play`.

step 4: autoplay Having the transcript of the game printed to the terminal is great when we're running one game at a time, but when we really want to know which strategy is better, we'll want to take a sample of thousands of games. To do this, create `autoplay`, a function that does roughly the same thing as `autoplayLoud`. The difference is that instead of printing the transcript of the game to the terminal, the simulation is done "quietly".

To communicate the winner to any testing code we write, the `autoplay` function needs to return a value that indicates who won that play of the game. Have it return the value 1 if the first player (the strategy listed in the first argument) wins. It should return 2 if instead the second player wins. Return a 3 when it's a tie.

step 5: manyGames Let's now write code to compare strategies. Any single game is going to include a lot of luck, and possibly a big advantage for one side based on who goes first. Write a function `manyGames` to give us a much more reliable way to compare strategies. It should take two strategies (like `autoplayLoud` and `autoplay`) as parameters along with a third integer parameter `n`. It should then run the `autoplay` function `n` times. For half of those it should have the first strategy roll first, acting as Player 1 (or nearly half when `n` is odd). It then lets the second strategy roll first for the remaining half. It keeps track of wins and ties, and prints a summary at the end. This summary should look like what's below:

```
Player 1 wins:  496
Player 2 wins:  503
Ties:           1
```

In our experience with this project, when you write `manyGames` you might introduce bugs, but the code could still run to completion. To give you some confidence, run `manyGames(sample1, sample1, 1000)`. The output just above resulted from doing this with our solution. Though there is randomness here, your function's output should be similar to what you see above (nearly equal performance by both players, only a few ties) most of the time.

step 6: write a second strategy. Now we ask you to write a strategy. It should behave as follows:

If its current score is no more than 50, it should roll 30 dice. If its current score is between 51 and 80 inclusive, it should roll 10 dice. When its score is above 80, it should pass.

Write this code in the function definition named `sample2`. Experiment to see how it compares to `sample1`. You should find that in large samples, switching which strategy goes first, it consistently beats `sample1`. If you don't, you have an error somewhere.

step 7: improve. One thing for sure: any decent strategy will never roll when its score is already 100. Such a roll would risk going over and losing, and it couldn't possibly help. Unfortunately, some strategies might roll even in this situation. Write a function `improve` that adds this check to another function's strategy. That is, `improve` should take a strategy function as a parameter, and it should give back (with a `return` statement) a strategy function that encodes this improvement/tweak. This new strategy function should choose to roll the same number of dice as the original one fed to `improve` in all situations, except in the situation where the player's score is 100, it should always pass, regardless of what the given strategy would have done.

Following my template in lecture, this means you'll write the `improve` function like this:

```
def improve(strat):

    def better_strat(myscore, theirscore, last):

        # code that uses strat but maybe acts differently

    return better_strat
```

The lines of code within `better_strat` can reason about `myscore`, `theirscore`, and so forth, just as any other strategy might. But it will want to call `strat` in certain situations to see what strategy `strat` would do in that same situation. Because of the nested function definitions, you can use `strat` within `better_strat`.

This is another great use of higher order functions within this project. You may need to delay starting this coding until we've covered higher order functions more completely in lecture. If so, perhaps skip to the next step. When we have covered passing functions as parameters and returning functions back this will be a great way to test your understanding of those concepts.

step 8: compete. Now it's time for the big project, writing your own strategy. This one is completely up to you. It can work any way you want, but your goal is to make the strategy as good as possible. Be creative. Try lots of things. Tweak the strategy a lot---changing a couple little details can have a big effect.

To complete this step of the assignment, we only want you to submit one strategy. Develop several strategies if you like, but we ask you to choose one that you officially submit. That one you choose, call it `myStrategy` . You can keep the other strategies, named differently, in your `greed.py` file. We will be happy to take a look at that other work and see all the things you tried.

Note that, because of the luck factor, the difference between a decent strategy and a really good one could be appear as a small effect on the win percentage. Say one strategy *A* has a 60% win rate against a given opponent *X*, while another *B* has a 58% win rate against the same opponent *X*. Then *A* might win very decisively when pitted against *B*. You should try writing several strategies that take substantially different approaches and then see which one performs best.

But, again, in the end, you should have one strategy that you are willing to stand behind, written in the `myStrategy` function of the program. The strength of that strategy will be a factor in your grade for this assignment, and it is also the strategy that will be entered for you in the class tournament. Remember, the resulting function might not be extremely long, but that doesn't mean you shouldn't put a lot of work into figuring out exactly what will work best.

We have a few technical rules restricting how the strategies can behave. The function must compute the next turn *from scratch* each time it is called. (For example, you cannot write to a file to save computation from previous calls.) Similarly, you cannot compute moves in some other way and attempt to write out all possibilities in your function definition. The strategy must be pretty fast. Running `pr1testing.testStrat(myStrategy, 10000)` (described below) should take no more than 30 seconds. How fast things actually take obviously depends on the particular computer they are run on. This time limit, in our experience, is not a hindrance to most things you might reasonably do. If you're worried about your strategy taking too long, talk to us.

Everything you write should be well-documented, and this is particularly true of your strategy. Write comments that explain what it is doing and (if not self-explanatory) why that is a reasonable thing to do.

Several tools have been provided to help you write your strategy. The `pr1testing` module, contained in `pr1testing.pyc` , contains ten strategies of varying quality for you to test yours against. You are not, however, allowed to look at `pr1testing.pyc`. (It will just look like gibberish anyway, but trying to decipher the gibberish is forbidden.)

Note that there is already an `import` command for this testing file already in the `greed.py` file. You can run a strategy by referring to, for example, `pr1testing.test6` (for the sixth of the eleven test strategies). That means that you could run

```
>>> manyGames(myStrategy, pr1testing.test8, 10000)
```

to see a sample of 10000 games between your strategy and the eighth test strategy. There is also a function that will run consecutive tests against all eleven of the test strategies. If you enter

```
>>> pr1testing.testStrat(myStrategy, 10000),
```

you will see the result of eleven matches---a match playing 10000 games of **Greed** using `myStrategy`---each match against each of the eleven test strategies. You can of course also watch games between your strategy and the test strategies (or two of your strategies, two of the test strategies, etc.) using the `autoplayLoud` function.

You should try to get your strategy to beat as many of the test strategies as possible. Your strategy will not be judged by the margin of victory over the tests, just whether it beats them. (For example, it is much better to have a strategy that consistently edges out wins over all eleven tests than one that decisively beats six of them but loses to the other three.)

We plan to hold a tournament between the strategies submitted by the class. In the tournament, will pit people's submitted `myStrategy` functions against each other in one-on-one matches. Each match will involve large runs of `manyGames` and we'll pick winners based on the aggregate results of a match. This tournament has been a lot of fun to watch in the past, and is meant to be low pressure: the outcome of the tournament will not affect your grade. (Though there will be prizes for the top tournament winners...)