Project 1: A Game of Greed

In this project you will make a program that plays a dice game called Greed. You start only with a program that allows two players to play it against each other. You will build tools that let computer players play the game against each other, as well as tools that help you figure out how good different computer players are. Then you will write your own strategy function, which you'll use to compete against some test players I have written for you, as well as your classmates. Have fun!

Rules of the game

The game of Greed is a dice game between two players. The dice in question differ from regular dice in that they give numbers between 0 and 5, instead of between 1 and 6. The game begins with both players having a score of zero and proceeds in turns. The player whose turn it is can choose any number of dice to roll. They then roll those dice (all at once) and add the total of the dice to their score. If they go over 100, the other player is declared the winner. (A score of exactly 100 is acceptable.) Play then passes to the other player, who does the same. Play alternates in this manner until either one player goes over 100 and loses or a player chooses to pass. When a player passes (that is, rolls zero dice on their turn) the other player gets one more turn to roll, and then the game is over. If at that point neither player has gone over 100, the player with the higher score wins. (Ties are possible.)

Step 1: `experiment`

You should first work to understand the game. The included `play` function conducts a game between two human players, both typing their moves into prompts in the terminal. You should play several games with a friend (or just against yourself) to get a feel for the game. You should then look at the code for `play` and try to understand how it works. (It is purposefully not documented very well.)

Step 2: `autoPlayLoud`

We want to have programmed strategies play this game against each other, rather than just having humans play it. `pr1.py` contains a sample strategy, a function named `sample1`. This strategy, like all other strategies we will write, takes three inputs. Those inputs represent the current state of the game the strategy is playing. The strategy takes those inputs and outputs the next move (that is, how many dice to roll). The first input is an integer representing the strategy's current score. The second input is an integer representing the opponent's current

score. The third input is a Boolean indicating whether the opponent passed on their last turn, which would mean that this roll is the last of the game.

You should look at the sample strategy, `sample1`. It is very simple – 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 currently in the file.

Your job is to implement `autoplayLoud`, which should work similarly to `play`, except that it conducts games between two computer strategies instead of two people. It should still print a record of the game to the terminal, but instead of asking for moves it should take as input two strategies and use those strategies to compute the move at each turn. (The strategy in the first argument should be "player 1" and should go first, and the other strategy should be "player 2".) To test your function, try running `autoplayLoud(sample1, sample1)`, which should conduct a game between two players each using `sample1` as their strategy. An example of what this should print to the terminal is below. (It is not necessary to have `autoplayLoud` return a value.) You can look at the code for `play` to get a general idea of what it should 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.
```

Step 3: `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`, 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". The function should return a value that indicates the winner of the game. `1` should be returned if the first player (the strategy listed in the first argument) wins, `2` if the second player wins, and `3` if there is a tie.

Step 4: `manyGames`

We now want to be able 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. The function `manyGames` gives us a much more reliable way to compare strategies. It takes as input two strategies (the first referred to as "player 1") and an integer n. It then runs `autoplay` n times, with each player going first for half of those times (or as close as possible if n is odd). It keeps track of wins and ties, and prints a summary at the end as shown below.

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

You should implement `manyGames`. You can test it using `sample1`. (The output above comes from running `manyGames(sample1, sample1, 1000)` and is typical of what output from that function call should look like. Remember that there are random numbers involved here, so the output will be slightly different each time.)

Step 5: `another strategy`

Now you will write a strategy. It should behave as follows:

- If its current score is no more than 50, it should roll 30 dice
- If it's current score is between 51 and 80, it should roll 10 dice
- If the score is above 80, it should pass

This should be written in the function `sample2`. You should then experiment to see how it compares to `sample1`. (You should find that in large samples it consistently beats `sample1`. If you don't, you have an error somewhere.)

Step 6: `improve`

One thing that's very clear about any good strategy is that it will never roll when its score is already 100. (Such a role would risk going over and losing, and it couldn't possibly help.) Unfortunately, some strategies (like `sample1`) might roll even in this situation. Write the function `improve` so that it adds this check to another function. That is, `improve` should take a *strategy function* as input, and it should output *another* strategy function. The output should behave exactly as the input strategy would, except that why the player's score is 100, it should pass, regardless of what the input strategy would have done.

Step 7: `compete`

Now it's time for the big project, writing your own strategy. This one is completely up to you. It can work in 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 – change a couple little details can have a big effect. Remember also that the difference between a decent strategy and a really good one can look like a small effect on the win percentage. Say one strategy has a 60% win rate against a given opponent, while another has a 58% win rate against the same opponent. The first might win very decisively when the two strategies compete against each other. You should try writing several strategies that take substantially different approaches and then see which one performs best. 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 figure out exactly what will work best.

Technical rules on how the strategies behave:

- The function must compute the next turn from scratch each time it is called. (In particular, 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. The time limit is intended to not be a hindrance to most things you might reasonably do. If you're worried about your strategy taking too long, talk to me.

Everything you write should be well-documented, but 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 nine 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.) There is an import command already in the file. You can run a strategy by referring to, for example, `pr1testing.test6` (for the sixth of the nine test strategies). That means that you could run `manyGames(myStrategy, pr1testing.test8, 10000)` to see a sample of 10,000 games between your strategy and the eighth test strategy. There is also a single command, `pr1testing.testStrat(myStrategy, 10000)`, that will run consecutive tests against all nine of the test strategies. You can 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 nine tests than one that decisively beats six of them but loses to the other three.)

We will also hold a tournament between the strategies submitted by the class. Again, in the tournament all matches will be large runs of `manyGames` and only who wins the match will matter to the outcome. The outcome of the tournament will not affect your grade. (Though there might be some small prizes…)