CS3101-1 Python, Fall 2014: Optional make-up problems

Daniel Bauer

Total points: 12
Due date: Sun, Oct 26th 23:59pm

This is a set of voluntary make-up problems that you can complete to earn back points lost in previous homework problems. You can decide to work on none, one, or both of the problems. As usual, please submit your solutions to your Drop Box on Courseworks.

Part 1 (6 points) - `schoenfinkeled` decorator

Write a function decorator that allows to *Schönfinkel* a function[1]. Schönfinkeling is a transformation that modifies a function with n parameters so that it can be called as a chain of n functions each taking a single parameter.

For instance,

```
@schoenfinkeled
def sumthree(a,b,c):
    return a + b + c
```

would define a function with the following behavior:

```
>>> sumthree(1,2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: new_fun() takes exactly 1 argument (3 given)
>>> sumtwo = sumthree(1)
>>> sumone = sumtwo(2)
>>> sumone(3)
6
```

or simply

```
>>> sumthree(1)(2)(3)
6
```

Here are some hints:

- Instead of explicitly checking if a function accepts only a single argument, you can handle the `TypeException` that is raised when you try to call the function with the wrong number of arguments.
- Your modified function should return another function (with $n-1$ arguments), to which you can apply the *Schoenfinkel* operation recursively.

[1] Named after the Russian logician Moses Schönfinkel (1889-1942). This technique is also called *Currying* after the American logician Haskell Curry (1900-1982) who rediscovered it later.

Part 2 (6 points) - Rock, Paper, Scissors Revisited

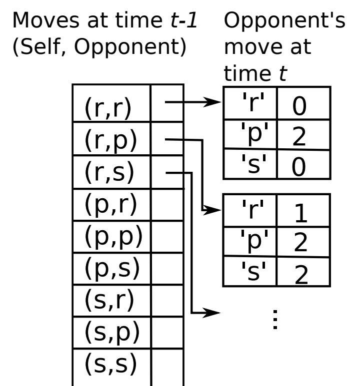
In this problem we develop an artificial intelligence for the Rock, Paper, Scissors game from problem set 4. You can base this extension on your own code, or use the sample solution on the course website.

The AI keeps a limited history of previous gestures (its own and its opponent's) and uses this information to predict the opponent's behavior. It assumes that its opponents gesture at time t depends entirely on the pair of gestures shown at time $t-1$.

(a) Create a new class `AiPlayer` that uses `Player` as a base class. Add a new method `learn(gesture)` to `Player` which is used to communicate an opponents last gesture. For the standard `Player`, this method does nothing. Change the game `main()` function to use `AiPlayer` and to communicate moves to both players.

(b) Override the `learn` method and the `play` method in `AiPlayer` to keep track of previous gestures played in the last two turns.

`AiPlayer` instances should keep a table, mapping a pair of gestures at time $t-1$ to a mapping from the opponents gesture at time t to a frequency count (i.e. how often the opponent played this gesture following the pair of gestures at $t-1$). For instance, the dictionary structure could look like this:



(c) Modify the `play` method to use the table from part (b) to select a gesture. Assume the opponent will play the gesture he used most often following the gestures in the previous turn. Then choose the gesture that would defeat him. If the table does not contain a pair of moves yet, choose a random gesture.

(d) Extra credit for any improvements to the AI's strategy.