MIT AITI
Python Software Development

Lab 06B: Object-Oriented Programming, Continued

In this lab, you will continue the implementation of your sports statistics application by designing multiple new classes to represent Teams and Matches. Please read the instructions carefully.

➔ **Copy your "lab06-2.py" to a new Python file called "lab06-3.py"**

The local web company that was interested in your program is convinced that you are on the right track. They want you to make certain to capture metadata about teams and matches next.

Part I – Object-Oriented Programming II – Your First Classes

While we provided the skeleton for the `Player` class in the previous lab exercise, we are leaving the initial implementation of subsequent classes completely to you, though we do specify certain behaviors.

1. Create a class called `Team`. The class should contain the following attributes:
    - `name`
    - `league`
    - `city`
    - `country`
    - `managerName`
    - `conference` – This refers to the international governing body under which the team plays, e.g. UEFA, CAF, ICC East Asia-Pacific, etc.

    All of these attributes should be required in the `__init__` function. Note part of the beauty of Python – because types are dynamic and inferred, none of the code of `Player` class needs to be changed even though `Team` is now a class and not a `string`.

2. Add a `list` or other data structure for teams in the module: `__teams` and initialize it with `Team` objects appropriate to the players you defined in the previous lab. You will need to add this earlier in the module than the `__players` definition. Implement a function `addTeam()` analogous to `addPlayer()` from the previous lab.

3. Change the player objects to use the teams you have just defined rather than the strings from the previous lab. Ensure that you are re-using the teams you have already defined by fetching them from `__teams` rather than redefining them.

4. Create a class called `Match`. The class should contain the following instance attributes:
    - `homeTeam`
    - `awayTeam`
    - `date`
    - `homeScores`

- `awayScores`

of note are `homeScores` and `awayScores`, which should be Python `dict` objects containing a map from `Player` full name (`firstName` and `lastName`) to a tuple (player,score). **Note:** this is not a clean or efficient way to do this, but using objects as dictionary keys in Python requires them to support specific properties which we won't go over just yet. Extra credit for those who improve this implementation, either by using `Player` objects as keys, or through some other means.

5. Implement functions `def homeScore(self)` and `awayScore(self)`which return the sum of the player scores in homeScores and `awayScores`. Naturally each should return 0 if no one on the team has scored.

6. Implement a function `def winner(self)`which returns the `Team` which has the higher score, as determined by the sum of the scores of each team.

7. Implement functions `def city(self)`which returns the city based on the city of the home team, and `country(self)`which performs an analogous function.

8. Implement a function `def addScore(player,score)`which:
    - Determines which team the player belongs to using his/her .team attribute
    - Determines whether that team is the home or away team
    - Adds that score to the appropriate `dict` if the player is not in it, or adds the score to the player's existing score if he/she does.

9. Create a few `Match` objects using the `Team` and `Player` instances you created earlier, storing them in a data structure `__matches`, and making sure to add scores on either side. Test the functionality of the functions you have defined.

10. Now that you have scores associated with the each match, storing a duplicate of the data in the player records is unnecessary (though allowable, depending on the structure of your data model). Remove `__scores` and `addScore()` from the `Player` class, and reimplement `totalScore()` and `averageScore()`using the attributes from `Match`. This can be done as follows:
    - Iterate through the matches looking for those in which the player's team has played
    - Get the `dict` `homeScores` or `awayScores` and fetch the score with the player's full name as the key.

11. Reimplement the functions from 3 of the previous lab's first part:
    - `def highestScore()` - returns the highest score by anyone in the system in a `tuple`: (player name, team, date of score, score).
    - `def highestScoreForPlayer(player)` - returns the highest score by the supplied player in the same `tuple` as above, or `None` if the player is not in the system.
    - `def highestScorer()` – returns the name of the player with the highest scoring sum, i.e. the total of all the goals/runs/etc. stored in the system.
    - `def highestAverageScorer()` – returns the name of the player with the highest average, i.e. the total of all the goals/runs/etc divided by the number of matches.

12. Reflect on, and write a few sentences about how this problem might be approached if you could only program in an imperative style. What data structures, functions, and other supporting elements would need to be added?

13. Consider the one-to-many relationships of teams and players – a player plays for a single club team. What are the advantages and disadvantages of making the team an attribute of a player rather than a list of players an attribute of the team?

    In reality, players can play for more than one team, typically a national team and a club team. How does this many-to-many relationship change the dynamic above? Which structure – lists of players belonging to teams, or lists of teams belonging to players – is better? Why?