◆ A Sports League

The task in this exercise is to create a `SportsLeague` class which has a number of teams, each of which has a certain strength, that can organize matches between any two teams, and run a tournament.

A test script called `"runtest.py"` is already written to help you when you're writing your code.

Here are the details of the tasks that you need to do to complete the exercise:

1. Write a class called `SportsLeague` whose constructor takes a single argument, a string denoting the league's name. The constructor should store this name in an internal attribute, and also initialize an empty dictionary internally. The constructor should check that the argument is a string (otherwise it should raise a `TypeError`) and that it is non-empty (otherwise it should raise a `ValueError`).

```
In [1]: league = SportsLeague("The Calvinball World League")

In [2]: league.teams
Out[2]: {}

In [3]: league.leaguename
Out[3]: 'The Calvinball World League'

In [4]: SportsLeague(43) # → TypeError
In [5]: SportsLeague("") # → ValueError
```

2. When the built-in function `len` is called on an object of this class, it should just return the length of the internal dictionary:

```
In [6]: len(league)
Out[6]: 0
```

3. Write a method called `addteam` that takes three arguments: a team name, its "strength", and an optional third argument called `force` which should be a `bool`. The `force` argument must default to `False`. The method should *not* check the type of the arguments directly, but it should convert them to a `str`, a `float` and a `bool` respectively (let Python check by itself whether the conversion is possible and raise an error if it isn't). *After* that, the method should check that the name is non-empty and that the strength is strictly positive, and raise a `ValueError` otherwise. Then, if `force` is `False`, it should check if the `name` is already in the internal dictionary, and raise a `KeyError` if it is. Finally, it should then add the team to the internal dictionary:

```
In [7]: league.addteam("Shorts",  0.9)
In [8]: league.addteam("Talls",   1.2)
In [9]: league.addteam("Olds",    0.5)
In [10]: league.addteam("Mutants", 2.0)

In [11]: league.teams
Out[11]: {'Shorts': 0.9, 'Talls': 1.5, 'Olds': 0.5, 'Mutants': 2.0}

In [12]: len(league)
Out[12]: 4

In [13]: league.addteam("A", "B") # → fail attempting to convert "B" to a float
In [14]: league.addteam("", 2.0)  # → ValueError (empty name)
In [15]: league.addteam("A", 0.0) # → ValueError (non-positive strength)
In [16]: league.addteam("Talls", 1.5) # → KeyError (already there)

In [17]: league.addteam("Talls", 1.5, True) # force=True → overwrite

In [18]: league.teams
Out[18]: {'Shorts': 0.9, 'Talls': 1.5, 'Olds': 0.5, 'Mutants': 2.0}
```

4. Write an auxiliary function (outside of the class, not a method!) called `dictranking` that takes a dict as an argument (no need to check the argument) and returns a list in which the keys of the dictionary appear sorted according to the ranking

of the values, from the largest to the lowest. For example:

```
In [19]: dictranking({"a": 2, "b": 5, "c": 3})
Out[19]: ["b", "c", "a"]
```

Hint: use `sorted` with a `key` argument, using a `lambda`; we've actually seen this exact situation in class…

5. When a `SportsLeague` object is printed (with `print` or in the console REPL), it should print the league name, followed by a colon and a newline, then by the teams, one team per line, each with its strength printed. The teams should be ranked by strength with the strongest at the top, using the `dictranking` function. The result should look like this:

```
In [20]: league
Out[20]:
The Calvinball World League:
 * Mutants: 2.0
 * Talls: 1.5
 * Shorts: 0.9
 * Olds: 0.5
```

Hint: build the output string one piece at a time, starting from the title and growing it line-by-line with `+=`. Remember the newlines after each team line.

6. Write a `_strength` method that takes two arguments: a team name and `homeboost`, a `bool` that determines if the team is receiving a strength boost by playing at home. The returned value should be just the strength as stored in the internal dictionary if `homeboost` is `False`, but it should multiply it by 1.2 if `homeboost` is true:

```
In [21]: league._strength("Olds", False)
Out[21]: 0.5

In [22]: league._strength("Olds", True)
Out[22]: 0.6
```

7. Write a `match` method that takes the name of two teams as the arguments. Here is how it should work: it should use the `_strength` method to get the teams strengths, giving the home boost to the first team. Then for each team it should compute a maximum number of points that it can make, by computing a "relative strength" and multiplying it by 5, and finally adding 1: for example, for team 1, the formula to use is

$$m_1 = \left\lfloor 5 \times \frac{s_1}{s_1 + s_2} \right\rfloor + 1$$

where s_1 and s_2 are the two strengths obtained above, and $\lfloor \cdot \rfloor$ denotes the integer part of the number (the `floor` operation). The formula for team 2 is analogous. You should then extract at random a score for each team between 0 and its value $m_{1/2}$ (included), using `random.randint`, and return the two scores. Remember that you need to `import random` at the top of the file. Also, while developing, for your own sanity set a seed! You can do that at the top of the file too. Here is an example of the method in action (your results will differ, of course):

```
In [23]: league.match("Olds", "Mutants") # max scores are 2 and 4
Out[23]: (1, 0)

In [24]: league.match("Olds", "Mutants")
Out[24]: (1, 1)

In [25]: league.match("Olds", "Mutants")
Out[25]: (0, 2)

In [26]: league.match("Mutants", "Olds") # max scores are 5 and 1
Out[26]: (4, 0)

In [27]: league.match("Mutants", "Olds")
Out[27]: (3, 1)
```

8. Write a `tournament` method that takes no arguments. Here is how it should work: there should be two matches for each pair of teams in the league, where one of the two teams has the home advantage each time. In other words, each team

should play twice against all the other teams, once at home and once not.

For each match, you should determine from the scores if the first team has won, or the second one has won, or if the match was a draw. You should keep a dictionary that keeps track, for each team, of how many wins, draws and losses it has had: the dictionary should have the team names as keys, and a list of three integers (wins, draws, losses) as the values. Initialize everything to zero and update it after every match. At the end, return this dictionary.

```
In [28]: results = league.tournament()

In [29]: results
Out[29]: {'Shorts': [1, 2, 3], 'Talls': [2, 1, 3], 'Olds': [2, 2, 2], 'Mutants': [4, 1, 1]}
```

[extra] Before returning, put some extra check in there: each team should have played $2(n-1)$ matches, where n is the number of teams, therefore each associated list should sum to that value (in the above example, each list sums to 6). Write an assertion that this is true. Try to do that in one line of code, using `all` and `map` and a `lambda`.

9. Write two functions (outside of the class, not methods!) called `getpoints` and `analyze_results`.

   The `getpoints` function takes a single argument, a list of three ints representing wins, draws, losses. It outputs the total points for a team, computed by assigning 3 points to each win, 1 point to each draw, 0 points to each loss.

   The `analyze_results` function takes as input a dict of the kind given by the `tournament` method, and it *prints* the teams and their points, in ranked order from the best to the worst, as in this example:

   ```
   In [30]: analyze_results(results)
   Tournament results:
   Mutants 13
   Olds 8
   Talls 7
   Shorts 5
   ```

   Ignore all issues that may arise if two or more teams have the same number of points — the ranking can be arbitrary. The method returns `None`.

   Hint: Here are a couple of ways to do this. 1) Use `getpoints` to create a new dictionary, in which the keys are the team names and the values are the scores. Use either a comprehension or `map` to do this (for the second option, you may want to look at the help for `dict`). With this second dictionary, you should determine the ranking of the teams using the `dictranking` function. 2) Determine the ranking with `sorted` and `getpoints`, then use `getpoints` again in the printing.

10. Write a method of `SportsLeague` called `odds` that takes a single argument, a positive integer `n`. The purpose of this method is to perform `n` tournaments and collect data. More precisely, for each team, it should estimate the probability of winning the tournament, that of being ranked second, third and so on. It should return this data as a dict of lists, as in this example:

    ```
    In [31]: league.odds(100)
    Out[31]: {'Shorts': [0.11999999999999998, 0.2900000000000001, 0.38000000000000017,
    0.21000000000000005], 'Talls': [0.2900000000000001, 0.4300000000000002, 0.21000000000000005,
    0.07], 'Olds': [0.03, 0.04, 0.24000000000000007, 0.6900000000000004], 'Mutants':
    [0.5600000000000003, 0.24000000000000007, 0.17, 0.03]}
    ```

    In the above example, the “Shorts” have an estimated 12% chance of winning the tournament, a 29% chance of arriving second, 38% or arriving third, and 21% of arriving last (the total is 100%, as it should). Also, the probability of winning the tournament is 12% for the Shorts, 29% for the Talls, 3% for the Olds and 56% for the Mutants (again the total is 100% as it should). As an extra challenge, you may `assert` that the totals sum up to about 100% (allowing for floating point errors) for each team and for each final position. Can you do it in just 2 lines of code?

Additional extra challenges and proposals for extensions

11. The function `analyze_results` and the method `odds` ignore have the issue that if two teams have the same number of points at the end of a tournament, their ranking is arbitrary. What if we wanted to disambiguate that, for example saying that in case of ex-aequo the team that has the larger number of wins prevails? Try implementing that. What about if they

also have the same number of wins? Maybe we want to use the total scores, but for that we need to collect more data from the `tournament` method. Try implementing that.

12. Related to the previous point, maybe we want to collect even more data from the `tournament` method, e.g. number of home wins, and so on. Maybe it would be a good idea to create a dedicated class for doing that? Try to think about what would be a good design for this kind of data collection, with the idea that you probably also want to analyze it in various ways.

13. Of course, a single “strength” property is hardly enough to characterize a sporting team. Maybe you could thing about implementing a more realistic setup, with multiple attributes that a team can have, and a more sophisticated scheme for simulating the outcome of a match. Again, at some point it may be worth putting the team data in a class.