Lab Practical 3: Abstract Data Structures and Objects

This week's practical will have you working with a larger movie dataset from the Open Movie Database (http://www.omdbapi.com/) in the form of JSON data files. You will write a python class to store and handle the movie data. With these Movie objects, you will create a simple movie recommendation system which will recommend a movie from the dataset based on the similarity to a movie that you provide it.

Setup

If you have not set-up your account with Anaconda, please follow the instructions in Lab Practical 1 to do the set-up.

Part 1 Get some Movie Data

You'll find a `movies.json` file in the `data.zip` for this week's practical. It contains many more movies than last week's lab practical. Download and unpack the data. It is always helpful to first take a look at the structure of the data. When the data file is considerable large, on a unix-based machine, you can have a preview look at the first few lines of the file with the command `head`.

On the command line, type:

```
head data/movies.json
```

As a JSON file, what data structure(s) is the movies data stored in?
Given this data structure, how would you step through the list of movies?

Part 2 Load the Movie Data using JSON

In this part, you will load the movie data from the `movies.json` into a data structure in Python.

a. Open a new Jupyter Notebook in the same directory where you unpacked your `data.zip` file.

b. Using the `json` module's `load` function that we covered in this week's lecture, to load the movie dataset from `movies.json` into a list of dicts data structure like we used in Lab 2.

c. Calculate how many movies you loaded from the movies.json, and print it out.

d. Print out the title and year for each movie in the dataset, to see a listing of the movies that are in the dataset, for example:

```
Star Wars (1978)
```

Part 3: Create the Movie class

Now you will create a Movie class to convert the movies in your dataset into a collection of objects.

a. Using the examples from this week's lecture, create a class in python called Movie with a constructor (i.e. `__init__()`) function.

   Add arguments to the constructor to store the following information from the movies dataset as class attributes to the Movie class:

   1. *Title*
   2. *Year*
   3. *Director*
   4. *Actors*

5. *Rating*
6. *Genre*
7. *Choose 1 or 2 other attributes of your choosing from the dataset to store in the object.*

Note that you do not need to store all of the data from a movie record in your Movie record.

b. Test your Movie class by instantiating a single Movie object from the first data record from the movies dataset (i.e. the first dictionary in the list of dicts). Print out the movie's Title and Year using the movie's attributes.

c. Now it would be convenient to make the Movie object compatible with the python `str()` function. Using the example from lecture this week, overload the `__str__` method in your Movie class. Have your `__str__` method return a string containing the *Title*, *Year*, and one other attribute of your choosing.

   Instantiate a new Movie object from a data record from the movies dataset, and test printing out the Movie record.

d. Now adhering to the object-oriented concept of *encapsulation*, add a new method to the Movie class which tests whether an actor starred in the movie or not. You may want to look at Lab 2 at the `find_actors_movies()` function as it has *similar* functionality.

   Here is the method's signature for you to use:

   ```
   def actor_in(self, actor_name):
   ```

   Have `actor_in` method return `True` if `actor_name` starred in the Movie, or `False` if not.

   Instantiate a new Movie object from a data record from the movies dataset, and test the `actor_in` function with an actor who starred in the movie and with an actor who did not.

e. With your new Movie class, use a *list comprehension* to create a list of Movie objects from the *list of dict* of the movies dataset.

f. Using your list of Movie objects, count the number of Movies that the actor "Harrison Ford" starred in. You may use a for-loop to count the movies. Or, if you want a challenge, use a *list comprehension* to apply the `actor_in` method on the list of Movie objects, and count the number of `True` values in the resulting list.

   Hint: Take a look at the methods of Python lists:
   https://docs.python.org/2/tutorial/datastructures.html#more-on-lists

   Is there a method that is useful for you?

Part 4: Movie Similarity

Now that you have a Movie class, we will define an simplistic algorithm to calculate our similarity score between two Movies. We will define the similarity score such that the higher the similarity score, the more similar both of the Movies are to each other. Recommendation systems often use statistics and data mining to define the similarity of two pieces of data with each other, but for this lab we will keep our scoring method simple.

a. Add a method to your Movie class that takes another Movie object (`other_movie`) as the argument and it will return the similarity score as an integer. Here is the method definition for you to use:

   ```
   def similarity(self, other_movie):
   ```

b. Let us define the similarity of two Movies as an integer point system. Have the similarity function accumulate the score as:

   +1 point if the two movies have the same Rating

   +1 point if the two movies have the same Director

   +5 points if the two movies were released in the same year,
   or +4 points if they were released within 1 year (1980 and 1981)
   or +3 points if released within 2 years,
   and so forth until 0 points for movies released beyond 4 years

+ 1 point for each Genre type that overlaps

Return the total points for the similarity of two Movie objects.

Implement this scoring scheme to start, but feel free to come back later in the lab to refine and add more criteria (i.e. overlap with Actors, or other attributes).

c. Before you test your function, you will have to re-convert the movie dataset's list of dicts to Movie objects. *Before you do that on the whole dataset*, convert two data records to Movie objects and test your similarity score on them.

   Does it calculate the score that you expected? It should calculate the same score regardless with of the two Movie's similarity method you called. For example, for two Movie objects `m1` and `m2`:

   ```
   m1.similarity(m2) == m2.similarity(m1)
   ```

d. Now that you tested your `similarity` function, create a new list of Movie objects from the movies dataset. Using the first Movie in the dataset, use a *list comprehension* to get a list of similarity scores of that Movie to all of the Movie objects.

e. Find the maximum score from this list. *Hint*: check out the `max` function - https://docs.python.org/2/library/functions.html#max).

f. Find all of the Movies with this maximum score and print their title. Use the fact that the list of similarity scores is in the same order as the Movie objects.

   What Movie(s) have the maximum score?
   Did you find the original Movie you compared the others against?
   How would you find the other highest scored Movies?

Part 5: Build a Movie Recommender

Now that you have a method to compare two Movies with each other, let's now create an object that can do the recommendation for us.

a. Create a new class called `MovieRecommender`. Have the `__init__` constructor accept a list of Movie objects as it's argument. The MovieRecommender will use this list of Movies to pick the most similar one to recommend.

b. Add a method to MovieRecommender called recommend, it will accept a Movie object (`movie`) as an argument and it will return a recommended Movie from it's internal movies list.

   ```
   def recommend(self, movie):
   ```

c. Use the code that you wrote in **Part 4d-f** to implement the recommend function. It will calculate the similarity scores of the submitted movie to all of the stored list of Movies.

   Have the recommender find the Movie with the maximum score and return it. If there are more than one Movie, then pick randomly among those with the highest score and return that movie.

d. Now, it would be useful to also return the similarity score with the recommended Movie.

   One way to return both the score and the Movie object, is to return a *tuple* of the two pieces of data, as in:

   ```
   return (recommended_movie, similarity_score)
   ```

   This means that you will have both pieces of information returned from the recommend function.

   Another way to return the similarity score, is to add the score as an attribute to the Movie object in the recommend function, as in:

   ```
   self.score = similarity_score
   ```

   Or, you can always re-calculate the similarity between the original and the recommended Movie.

e. In the `data.zip` for this lab, you will find another JSON file `movie_queries.json.` It contains a set of new Movies that are not in the original `movies.json` dataset. For each Movie contained in `movie_queries.json`, print out the queried Movie, the recommended Movie, and the similarity score.

   Re-run the queried Movies through your MovieRecommender, d*id the recommended movies change?*

Congratulations you have now created a simple Movie Recommender!!!!

If you have more time, go back and refine your similarity function, and re-run the queried movies through your MovieRecommender. *Did the results change?*

Lab 3 Challenge: Pickle It!

Now that you have been experimenting with your MovieRecommender in the Jupyter Notebook, you can now create a python script to run the Recommender on the command-line.

However, reading in the movie dataset from JSON and creating Movie objects can take a long time. We will use the pickle module to optimize loading the MovieRecommender from the Jupyter Notebook.

a. In the Notebook, use `pickle.dump` to save a .pkl file of your MovieRecommender object. Because the MovieRecommender contains the list of Movie objects, everything you need should be saved to the .pkl file.

b. Now using the lab3-movie-recommender.py template, add the following code from your Notebook:

   1. Your up-to-date Movie and MovieRecommender class definitions (remember these do not export with the .pkl file).

   2. Code that loads the `movie_queries.json` file and parses it into Movie objects.

3. Code that gets a list of recommended movies and their scores from MovieRecommender

c. Instead of printing out the recommended Movies and their scores, output the results in a new JSON file `movie_recommendations.json`. Devise a data structure where you can return the original queried Movie, it's recommended Movie, and score.