OMDB DOCUMENTATION

Author: Casey O'Rourke

Contents:

A. Running the program and Requirements

The program is written entirely in Python and needs to be called from the command line. The arguments are:

1. input csv consisting of the movie,year
2. output_file_name.txt for writing the results
3. (optional, default=IMDB Rating) key to sort by
    - Supported keys (case sensitive):
        - imdbRating
        - imdbID
        - Year

It will be fairly easy to expand the supported keys, but these were the only ones I tested and did not receive an non-ascii character error. While the program is running, the console will print out the results of the most recent API call for a given movie, and its score.

Some examples of usage:

- Sort by default IMDB rating
    - *python omdb.py movies.csv output.txt*
- Sort by provided key:
    - *python omdb.py movies.csv output.txt imdbID*

In the same folder as the modules are the input files. The provided movies.csv is present, but I would recommend, given the time taken to run through the full list, starting with a smaller sample, provided as small_movies.csv to test for initial functionality. The program will create a log file to record any instances of a movie not being found (see section C for more details).

Requirements

The program was tested with Python 2.7 in both a Linux and Mac environment. **The program will not run in Python 3.0 or greater given the difference in print function calls**

in 2.7 and 3.0+. No additional libraries were installed. The program was also testing using Virtualenv and found to work. The built-in Python libraries used were:

- sys
- json
- urllib2

If your environment is missing any of these I would recommend using pip install: http://pip.readthedocs.org/en/latest/installing.html

The main folder also contains some example outputs obtained from running the program on my machine.

B. Design Considerations

There are two classes 1) omdb.py and 2) Movie.py. The omdb module is the main driver of the program, opening the CSV file and reading line by line, creating Movie objects for each line. The Movie.py module defines the Movie class and is responsible for parsing each line to make the ultimate call to the OMDB API, which is done inside the Movie class as well.

Given the time frame of two hours for completion and the instructions relating to API calls, I decide to go with a simple serial implementation of the program. The main bottleneck in performance, given the current input size, is the API calls. **Performance increases could be gained by through parallelization of the API call** and having multiple data structures store the movie objects to avoid concurrency issues. These improvements should only be made after handling most of the issues in section C, however, so as to avoid any premature optimizations that might compound the problems that currently exist.

Moreover, if we only cared about the IMDB rating, then we could sort the movies in constant time as we pull them from the API using a basic bucket sort given there are only 100 possible ratings. If the input size of movies were to increase into the thousands, this would be a good candidate for using MapReduce which would handle of the overhead of having multiple API calls and sorting.

The program currently stores the entire JSON response in the Movie object. I chose to do this to extend the potential uses for the program. As written, the program can sort based on a given optional key such as the "Year" or "imdbID".

Efforts were made to decouple the input processing from the API call. The processing of each line was one of the biggest problem areas given the variety of inputs. By decoupling the input processing, we can focus on dealing with special cases in this function alone.

C. Issues and Known Bugs

The two biggest challenges in this problem were dealing with special cases in the input and formatting a valid URL for the API call (these two issues being closely related). From an initial inspection of the input I found the following cases:

1. Spaces and other characters like quotation marks are in movie titles, which can result in malformed URLs

2. Some movies will not be found either because 1) It does not exist or 2) Movie title does not match provided year

3. Some titles have commas in them. Care needs to be taken in parsing the title and year

4. Non-ASCII characters can result in printing and encoding errors

Within the given time frame I attempted to handle most of these cases, albeit there is much room for improvement. In particular, for case 2, I simply output the title of the movie to a log file for later inspection. This will help maintenance of the program as unknown cases may arise and the user of the program can always check for errors. For case 4, I simply overlook this issue for now and remove the offending character or catch the exception. This is not ideal, but allows for the program to run with no problems. Eventually the Movie.__str__() method or the exception handler should take care of this special case.

The biggest issue that the currently exists in terms of the program failing is that there is nothing to handle issues with the API having errors in terms of a broken connection beyond ignoring the failure and moving on to the next movie.

Overall there are a number of items that need to be error checked in terms of valid inputs. The command line argument parsing is very crude as well. In addition, there is nothing checking the API results as well against actual scores. From a simple visual inspection, it was noticed that “E.T.” was noted as having a score of 2.2, while in reality its score is much higher.

D. To Do

- Movie.py
    - queryOmdbApi method: Handling failed API calls
    - queryOmdbApi method: Handling movie not found cases
    - processInput method: more robust url building from input string
- omdb.py
    - main method: More robust command line argument parsing. Use optparse library
    - main method: Check input file has valid contents

E. General Comments

I enjoyed this challenge, in particular with the time constraint. It forced me to prioritize features and other aspects of the program so I could get a working prototype, while programming in such a manner that would allow me to go back later to handle special cases. I ended up spending around 2.5 hours to get a working version. Another hour was spent adding a little more functionality to allow for sorting by additional keys and handling exceptions. Lastly I spent an hour typing up the documentation and tidying up my code.