Report on Quizzical

Ivor Walker

1 Overview

This submission provides all functionality described in sections basic, intermediate and advanced. It is a Python application that shows questions and a list of multiple choices answers obtained via the Open Trivia Database API to the user in a curses-based command line interface. The user has 30 seconds to select an answer or to play a chip. These chips (50/50, extra time, ask the host) gives the user an advantage in answering the question. If the user's answer is correct, the application adds to their score based on the difficulty of the question. This continues until the user quits or answers three questions incorrectly, at which point the game ends and the user's final score is displayed alongside a leaderboard of the top 10 scores and statistics on how long the session lasted.

2 Design

The entry point (main.py) initialises the view and game and starts the main game loop.

The View class (view.py) renders the game state to the terminal and contains all direct interactions with curses. I chose to seperate the view from the rest of the program to make it easy to swap out the view for a different view (e.g a GUI) without changing the rest of the program. The constructor initialises the curses screen and sets curses-related configurations. The view contains two methods that are the main ways of displaying information to the user: show_multiple_choice, which displays a question and multiple choice answers, and show_message, which just displays a message. There are other bespoke methods that deal with displaying information not displayable by these methods, such as showing the welcome form and showing the leaderboard.
Among other standard settings, in the constructor I enable raw mode to stop special character combinations (e.g Ctrl+C) from sending signals to Python or curses (e.g close the curses window) when recieving input. Instead, I handle these special character combinations within game.py.
Although my view supports character and line input, I primarily use character input to register special character combinations and to integrate with the timer as character input, unlike line input, can be interrupted at will (e.g when the timer expires).

The Game class (game.py) handles the game state. The constructor initialises the questions, timer, and leaderboard, resets the game state, and prompts the user for their username and bonus category. The start method is the method called by the entry point to start the main game loop. This runs the play_round method until the game is over.
Interactions requiring the user to choose one option from many are handled by the ask_multiple_choice method, which asks the user a question (via view.show_multiple_choice), waits for the user to input an answer, checks if the user's answer is a valid option and returns the user's valid answer. The show_message method shows a message to the user and waits for the user to press a key to continue. These two methods are the main methods that interact with the user, and have a variety of default options to make them easy to use in different contexts. The method that asks the user a trivia question (ask_question) is a wrapper around ask_multiple_choice that also shows the user's current status and adds "play a chip" as an option to the multiple choice answers. Future extra functionality can be easily added by creating new methods that wrap around these basis functions and modifying these default options.

KeyboardInterrupts are expected to be raised in the get_input methods and ConnectionErrors are expected to be raised in the fetch methods in the Questions class. Both these exceptions are handled within the play_round method, which catches these exceptions and starts the quit sequence. I bubble up these expected exceptions from these methods to their play_round method to properly end the round if the user quits or the API is unreachable.
To quit the application, the user presses Ctrl+C rather than a dedicated quit key, which raises a KeyboardInterrupt and starts the quit sequence. Users of console applications are used to pressing Ctrl+C to quit but may overlook an instruction to press a specific key to quit. For example, a common gripe with Vim is that Ctrl+C does not quit the application but requires inputting a quit command. One of the most popular questions on StackOverflow asks how to quit Vim [1]. This approach also sidesteps the need to escape the arbitrary quit key in any input (e.g if the user's username contains the quit key).

The Question class (question.py) represents a single question. The constructor takes in a JSON object from the API representing a question and stores the question's attributes. It unescapes HTML entities in the question and answers, combines the correct and incorrect answers into a single list of choices and shuffles it. The other methods (fifty_fifty, ask_the_host) modify the question in-place, which are called by the play_chip method in the Game class handling the user playing a chip. With the exception of the 'extra time' chip which calls a method of the Timer class, all chips call methods of the Question class. I chose to modify the question in-place to simplify playing multiple chips on the same question. Since the question object is discarded after the user answers it, modifying the question in-place has no effect on future questions.

The Questions class (questions.py) represents a list of questions, and fetches questions from the API. The constructor method fetches a new token from the API. I attach this token with any fetch request for more questions to ensure the API fetches unique questions. This approach to maintaining uniqueness of questions eliminates the need to check and remove duplicates from the list of questions, which reduces complexity and means all questions are usable - discarding a question because it is a duplicate would increase the number of API fetches required during a session. One disadvantage of this approach is its dependency on the API not sending duplicate questions, so I check whether any questions within a single fetch are duplicates and repeat the fetch if they are. This approach also makes it easier to swap APIs to one that cannot guarantee unique questions, as I can easily add a check for duplicates to the fetch method.
The fetching method fetches the maximum number of questions (50) from the API and adds them to the list of questions. Rather than fetching questions one at a time, fetching the maximum number of questions in a single fetch reduces the overall number of fetches performed during a session. Given the small size of the data (a question is maximum 1KB), the time taken to fetch 50 questions is not noticeably longer than fetching a single question. By minimising the number of fetches and instead storing the results of large queries, I reduce the time spent on fetching questions and the likelihood of rate limiting. This class also contains methods to interact with the list of questions, such as getting a random question and getting a question with certain attributes (i.e difficulty). The latter method is designed to be very open ended, taking a dictionary of any attribute and returning a random question that matches all attributes. This allows for easy extension

of the application to include more attributes (e.g category), at the cost of complexity of the method. I made this decision because I originally interpreted the "bonus category" requirement as a category that should come up more often, not a category that should come up equally often but be worth more points, so I thought it would be useful to have a method that could query questions based on any attribute to unify this requirement and the requirement to serve questions of a certain difficulty.

The Leaderboard class (leaderboard.py) represents the leaderboard of the top 10 scores. The constructor initialises the entries in the leaderboard, and there are methods to add a new score, sort and trim the leaderboard, and get user-friendly string representations of a single entry or a list of strings representing all entries. To show the leaderboard to the user, the result of the latter method is passed to the view's show_leaderboard method.

The Timer class (timer.py) represents a timer for measuring time elapsed (i.e measuring how long the user takes to answer). The constructor initialises the timer, and when a user is asked to answer a question the start method is called to start the timer. This records the time when the timer was started, and creates a seperate thread that starts a timing loop which recalculates the time elapsed since the loop started. Once a user has inputted an answer, the reset method is called which stops this loop.
I chose to run the timer loop in a seperate thread to prevent the timer from blocking the main thread. If I started a blocking timer before the user inputted their answer, it would only allow the user to enter an answer once the timer has finished. If I started a blocking timer after user input, it would only submit the answer once the timer had finished. By running the timer in a seperate thread, the user can input their answer and the timer can run concurrently, allowing the user to submit their answer and finish the question at any time.
The timing loop has a sleep which increases the time between elapsed time updates. A longer gap between timer updates reduces the number of iterations of the loop across the same time period, which reduces CPU usage at the cost of timer precision. I set the precision to zero decimal places (or to the nearest second) because the user has 30 seconds to answer, so the user needs precision to the nearest second.

3 Resources

This solution imports two external libraries and several standard libraries with Python to accomplish various requirements.
The external library 'requests' is used by the Questions class to fetch questions in a JSON format from the Open Trivia Database API. It does this by sending a GET request to the API's endpoint, which returns a response that this library can be converted to a JSON object or throws custom exceptions that I can handle. I used this library because of its simplicity - sending a GET request only requires one line of code, and converting it to JSON only requires one more line of code. Also, it is a standard library for fetching data from the internet so I could easily switch to a different API and still use this library if I needed to by changing the URL in the fetch method.

The other external library is 'curses', used in the View class to render the game state to the terminal. This library handles all inputs and displays all output in a terminal window. I

used this library because its methods are close to the terminal's low-level functions, which allows me to control the terminal window in a way that is not possible with higher-level libraries like 'print' (e.g move cursor around). This library is also cross-platform, so I can run the application on any operating system that has a terminal window. However, this library is not as user-friendly as higher-level libraries, so I had to write more code to achieve the same result and consult more documentation to understand how to use it. I ran into significant issues setting up KeyboardInterrupts and the only time I had to consult the internet for help was to solve this issue [2].

4 Evaluation

I tested this application by running it and playing it myself.

This solution meets all possible requirements and is fully functional. The application in its current form is easy to use and has a simple interface. The classes have lots of default options and basis functions for more functionality (e.g get_multiple_choice) have already been written, allowing existing functionality to be easily modified (e.g changing the amount of time the user has to answer) and future functionality to be quickly developed as wrappers around these basis functions. Documentation is present for all classes and methods, and the code is well commented.

However, some specific methods could be improved. The update_timer method in the Timer class could be replaced with a Timer class that uses a Listener-Observer pattern to update the timer. This would allow the timer to be updated without needing to check the time in the main loop, which would make the code more modular and easier to read. The filter method in the Questions class could be simplified and made less general safely by just specifying the requested difficulty.

References

[1] jclancy, StackOverflow. "How do i exit vim? "[Online]. Available: `https://stackoverflow.com/questions/11828270/how-do-i-exit-vim`.

[2] mhlester, StackOverflow. "How to raise keyboardinterrupt when inside curses. "[Online]. Available: `https://stackoverflow.com/questions/21123146/how-to-raise-keyboardinterrupt-when-inside-curses`.