Report on Quizzical

Ivor Walker

1 Overview

This submission provides all functionality described in sections basic, intermediate and advanced. It 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 answer is correct, the user gains score based on the difficulty of the question answered. The game ends when the user quits or answers three questions incorrectly. 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 or terminal interface) 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 methods that deal with displaying more specific information 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. Character input can be interrupted at will (e.g when the timer expires), unlike line input.

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, 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 enter. These two methods are the main basis functions that interact with the user, and have many default options to make them usable in different contexts and are easily wrappable to add more functionality. For example, 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.
To quit the application, the user presses Ctrl+C rather than a dedicated quit key to raise a KeyboardInterrupt. 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 text-editor Vim is that Ctrl+C does not quit but requires inputting a dedicated quit command. One of the most popular questions on StackOverflow asks how to quit Vim [1]. This approach sidesteps the need to escape the quit key in any input (e.g if the user's username contains the quit key).
KeyboardInterrupts are possible in the get_input methods and ConnectionErrors are possible in the fetch_new_questions methods in the Questions class. Both these exceptions are handled within the play_round method and starts the quit sequence. I bubble 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.

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, forms a list of choices the user can make, and shuffles it. 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 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. This approach relies on the API not sending duplicate questions, so I check if questions within a single fetch are duplicates and repeat the fetch if they are. This method makes it easier to swap to an API that cannot guarantee unique questions if needed.
fetch_new_questions fetches the maximum number of questions (50) from the API and adds them to a list of questions. Rather than fetching questions individually, fetching multiple questions in one fetch reduces the 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, I reduce the time spent on fetching questions and the likelihood of rate limiting.
This class contains methods to interact with the list of questions, e.g getting a random question or getting a question with certain attributes (i.e difficulty). The latter method takes a dictionary and returns a random question that matches all attributes. This allows for easy extension of the application to include filtering of more attributes (e.g category), at the cost of complexity. I decided on additional complexity 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. I thought a method that could query questions based on any attribute could be one method to solve 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. There are methods to add a new score, update the leaderboard, and get user-friendly epresentations of 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. 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 creates a seperate thread that starts a timing loop which recalculates the time elapsed since the loop started. Calling the reset method (e.g when the user answers the question) stops the timer.
I ran 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, the user could only enter an answer once the timer has finished. If I started a blocking timer after user input, it would only submit their 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.
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. The external library 'requests' is used by the Questions class to send a GET request to the API's endpoint, which returns a response that this library can convert to a JSON object or throw a custom exception that I can handle. I chose this library because of its simplicity and I could switch to a different API by simply changing the URL in the get 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 enables granular control of the terminal window (e.g moving cursor around). This library is cross-platform, so this application can be run on any operating system that has a terminal window. The library appears more readable as its design resembles other programatic UIs (e.g Swing). However, curses is not as user-friendly so I had to write more code to achieve similar results as higher-level libraries. For example, I ran into significant issues setting up KeyboardInterrupts with curses 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 testing all game states manually. I gave this application to my girlfriend to play who helped me identify usability issues. Constructing a test suite would be unecessary for this game because all possible states can be explored quickly.

The application is heavily reliant on user input and output which is difficult to test automatically.

This solution meets all possible requirements, is easy to use and has a simple interface. The classes have lots of default options and basis functions and basis functions (e.g ask_multiple_choice) so existing functionality can be easily modified and future functionality can be quickly developed as wrappers around these basis functions. Documentation is present for all classes and methods, and the code is well commented.

However, 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 would reduce calls to get_update() and CPU usage, and improve readability. The filter method in the Questions class could be simplified and made less general safely by specifying the requested difficulty only. Lots of design decisions in the Questions class were made with making swapping APIs in mind, but I did not implement this feature.

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`.