Final Exam

Programming, Algorithms & Data Structures

Program: MSc Business Administration & Data Science

Course: CDSCO2402E

Type: Home assignment

Submission date: December 20th, 2024

Examiner: Somnath Mazumdar

Author: August Bjerg-Heise (S152153)

Pages: 15 (excl. Appendix)

Characters: 29,614

Table of Contents

0. Abstract

The objective of this project is to develop a functional, interactive minimum viable product (MVP) of a trivia game that encourages families to engage in a shared activity during the Christmas holidays to reduce individual screen usage. The solution is implemented using a Python program that accesses the OpenTrivia Database API for dynamic question retrieval. The solution includes a simple graphical user interface (GUI) that allows the user to answer questions and track scores in real time as the players earn points by answering correctly. The program also employs an insertion sort algorithm for displaying the final leaderboard, chosen for its simplicity given the small set of players.

The resulting MVP fulfills the functional and non-functional requirements and is ready for user deployment. The program lives up to the expectations of flexibility and modularity intended to ease future development. Possible avenues of such development are identified, with short-term improvements including better handling of special characters and user-friendliness of the game launch, while long-term improvements include expansion of game modes and increased customizability with regards to question types and categories.

Keywords: Object-oriented programming; API integration; Graphical User Interface; Algorithmic Analysis

1. Introduction

In recent years, it has become increasingly common for families to spend their free time engaged with individual screens, rather than in each other's company. This is a disheartening development, particularly when considering the upcoming Christmas season. The purpose of this project is to address this issue by developing a fun and engaging trivia game that families can enjoy *together* during the Christmas holidays. By providing an activity that is both entertaining and inclusive, the project encourages a collaborative and festive atmosphere.

The approach to this project follows a top-down development process, beginning with an analysis of the functional and non-functional requirements. From this foundation, the system architecture is designed, to provide a high-level overview of the program's structure and interactions. The focus then shifts to detailed implementation, with particular attention to the design and functionality of key classes and functions. Finally, the project concludes with a critical reflection on the methodological choices, evaluating the program's outcomes and identifying areas for future improvement. This structured approach ensures that the project is both functional and adaptable, meeting its intended purpose while maintaining flexibility for further development.

Due to the resources available for development, the scope of this project is limited to a minimum viable product, which can be tested with select users during the upcoming Christmas. The goal is thus to achieve the right balance of functionality and speed-to-output while maintaining flexibility for further development, rather than reaching for the "perfect" game.

2. Requirements analysis

2.1. Functional requirements

The functional requirements of the trivia game define the foundational objectives and functionality of the system. In the case of this Trivia game, the key objective is to achieve user interaction, differentiation between children and adults, replayability, and accessibility.

Firstly, the program must present trivia questions in a structured and visually clear format, ensuring that both the question and multiple-choice options are distinctly displayed. Users are required to select one option as an answer for each question. Subsequently, the system must evaluate the selection against the correct answer, providing immediate binary (right/wrong) feedback to the user. Additionally, the system must track and display the user's total score during gameplay and at the end of the quiz, so the winner can be easily determined.

Secondly, the program must also allow adults and children with different fields of interest to play together, with appropriate adjustments to question difficulty to ensure fairness across varying age groups. Such adaptability promotes inclusivity and sustains interest among players with different levels of expertise. Furthermore, the system must encompass a wide range of question categories. This diversity minimizes the risk of players gaining an advantage through specialized knowledge in specific fields, such as computer science or sports, thereby ensuring balanced competition.

Lastly, the functionality should be accompanied by a simple GUI, to ensure ease of use for a non-technical audience.

2.2. Non-functional requirements

In contrast to the functional requirements, the non-functional requirements do not determine *if* the system works, but rather *how well* it works. The main objective here is to achieve a program that is family-oriented, runs seamlessly across different computers, and will continue to do so in the future. More specifically, this entails achieving “replayability”, usability, reliability, maintainability, flexibility, and portability.

“Replayability” means that it should be possible to play the game many times without seeing the same questions again and again. Achieving this requires access to a large and diverse database of trivia questions, ensuring each gameplay session remains unique and engaging. This functionality enhances the game’s longevity and provides players with a fresh experience each time they play.

Usability is a central consideration, as the system must provide a user-friendly and intuitive interface. Clear instructions should guide users through gameplay, ensuring that the game is accessible to individuals across different age groups. By emphasizing ease of use and accessibility, the system ensures that all users, regardless of technical proficiency, can engage comfortably and enjoy the experience, increasing the family-friendliness of the game.

Reliability is crucial for maintaining the user-friendliness and robustness of the game. The system must be designed to handle invalid inputs gracefully, ensuring that unexpected behaviors, such as a player failing to select an answer or entering improperly formatted data, do not cause crashes. Instead, appropriate error-handling mechanisms should provide feedback to users and allow them to proceed seamlessly. This capability enhances the overall stability of the system, ensuring a smooth user experience.

Maintainability is another essential quality, supported by adherence to modular and object-oriented design principles. By structuring the code into discrete and cohesive components, such as question handling, scoring, and the graphical user interface, the program ensures that updates or modifications can be implemented efficiently. This approach minimizes the risk of unintended side effects when changes are introduced, thereby extending the system’s longevity and adaptability.

Flexibility is vital for accommodating future expansions or enhancements to gameplay. The design must enable straightforward modifications, such as the introduction of new features, e.g., allowing users to select specific categories before the game starts, without necessitating significant restructuring. This adaptability ensures that the program remains relevant and capable of evolving alongside user needs and expectations.

Finally, portability ensures that the trivia game can be executed on a wide range of systems. The program must be compatible with any Python 3.x environment and rely on minimal external dependencies (such as uncommon or non-standard Python libraries). This ensures that the system can be easily executed on the average household computer, thus aligning with the accessibility goals of a family-oriented game.

2.3. Constraints

The constraints define the technical and operational boundaries within which the trivia game must be developed and executed. From a technical perspective, the program must operate efficiently without consuming excessive memory or computational resources. This ensures that the game can run seamlessly on an average household computer. Furthermore, the program should maintain compatibility across common operating systems, including Linux, macOS, and Windows, to maximize accessibility and portability. In terms of hardware and environment, the program is explicitly designed for execution on a single computer. This constraint aligns with the family-oriented goal of fostering group interaction and shared experiences, rather than isolating participants behind individual screens.

Finally, the constraints are guided by the course content and exam specifications. Given the course's focus on Python programming, the entire codebase must be implemented in Python. Additionally, as required by the exam guidelines, all code should be consolidated into a single Jupyter Notebook to streamline submission and evaluation.

3. System design

3.1. Architectural design

The system consists of interactions between three main components: 1) the user interface (GUI), 2) a Python program in a single Jupyter notebook, and 3) the Open Trivia database. A high-level overview of the system architecture is seen in Figure 1.

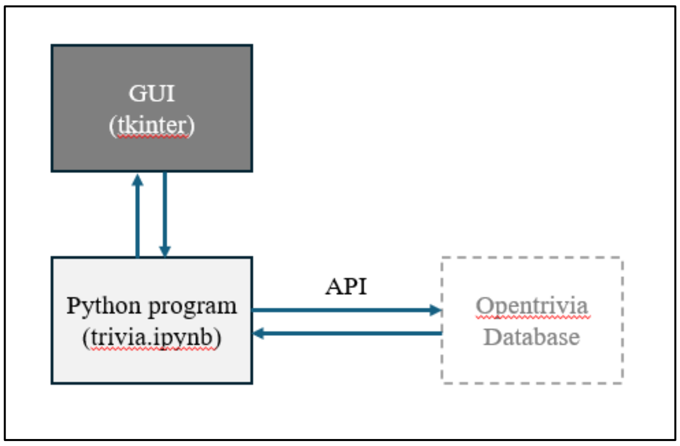


Figure 1: High-level overview of system architecture

The GUI, implemented using the Tkinter library, serves as the interactive interface for the users. It displays trivia questions along with multiple-choice options, records user inputs for answers, and presents the live leaderboard to track players' scores. The GUI communicates with the Python program by passing user input in the form of answers to questions and receiving player names & types, fetched questions, evaluated scores, and an updated leaderboard. While there is a wide array of libraries that could provide a more aesthetically pleasing GUI than Tkinter, using a standard Python library reduces the overall dependencies of the program, making it more in line with the identified requirements.

The Python program constitutes the core logic of the trivia game and is based on object-oriented logic with classes, functions, and a cohesive game flow. It is responsible for managing the overall functionality, such as storing player details (names, types, and scores), validating answers, and determining progress based on user inputs received from the GUI. Additionally, it interacts with the OpenTrivia Database through an API. The OpenTrivia Database is a publicly accessible repository of more than 4,000 trivia questions categorized by difficulty and type. The Python program fetches questions individually from this database. Relying on an external database of questions is chosen since it eliminates the need for a local database, reducing memory usage. However, it does introduce a dependency on this database, as well as the need for a stable internet connection. Based on the requirements specified in the previous section, this is deemed to be an appropriate trade-off.

When initializing the game, the user will first have to provide a few inputs within the coding environment, more specifically the number of players (any positive integer), their name (a string), and type (“child” or “adult”). This will trigger the opening of the GUI, which is responsible for the main interaction between the user and the Python program (see Figure 2).

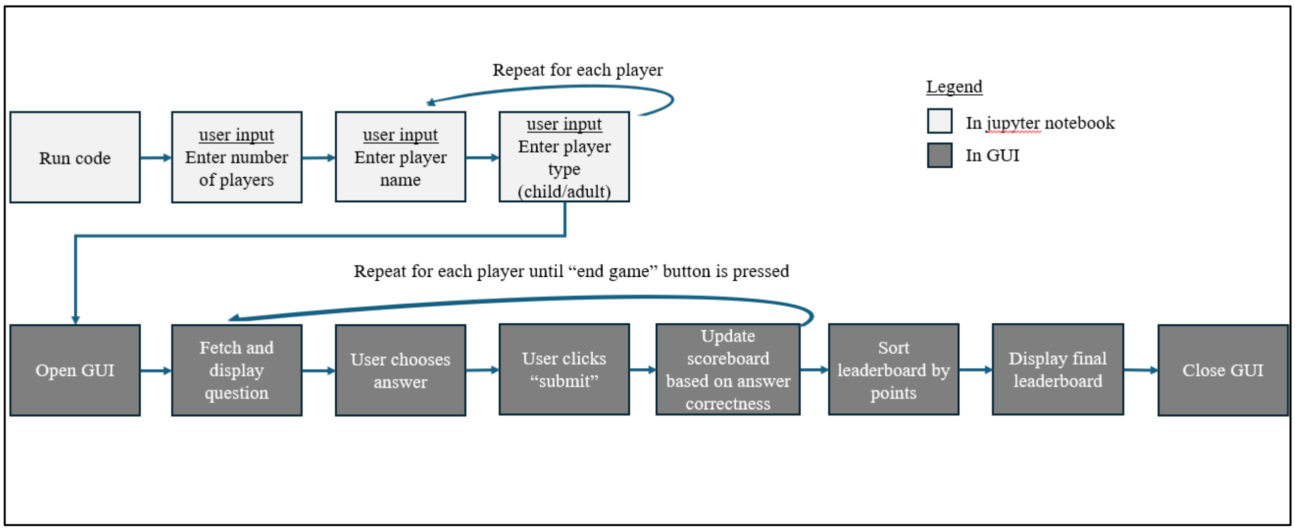

Repeat for each player
Legend
In jupyter notebook
In GUI
Run code
user input Enter number of players
user input Enter player name
user input Enter player type (child/adult)
Repeat for each player until "end game" button is pressed
Open GUI
Fetch and display question
User chooses answer
User clicks "submit"
Update scoreboard based on answer correctness
Sort leaderboard by points
Display final leaderboard
Close GUI

Figure 2: Game process flowchart

The user interaction is iterative in the sense that they are repeatedly presented with questions for each player, before answering them. When every player has answered, the first player is prompted to answer again. This loop continues until the user presses an "end game" button. This will trigger a sorting of the players by points in descending order, which is then presented to the user, after which the GUI is closed.

Insertion sort is chosen as an appropriate sorting algorithm due to its simplicity and applicability to short and nearly sorted lists. In this scenario the number of elements will be unlikely to exceed the size of a normal household (4-5 people), meaning that its worst-case time complexity of $O(n^2)$ does not pose a significant limitation. Furthermore, it is not unlikely that the list encountered is already close to being sorted, bringing the time complexity closer to the best case of $O(n)$. While other more complex algorithms, e.g. merge sort and quick sort which are more complex to implement and generally require larger datasets to reap the benefits of their logic, could show a slightly better performance, they are deemed to be excessive for this use case.

3.2. Module design

Following the requirements outlined in section 2, the code is built on a single module in a .ipynb file. This file contains two main functions: "get_questions()" and "insertion_sort()" and four main classes: "Question", "Player" (with subclasses "ChildPlayer" and "AdultPlayer"), "TriviaGame" and "TriviaGameGUI".

The get_questions() function fetches trivia questions from the OpenTrivia Database API based on the input parameters difficulty (easy/medium/hard), amount (number of questions fetched), question category, and type (multiple choice or Boolean). While each of these, except difficulty, is standardized throughout the program, the option to change them is included to ease the expansion of functionality in future development. It constructs a dictionary of the parameters and sends an HTTP GET request using the requests library. The response is validated for errors, and the question data is parsed into a Question instance using the from_api_response method. If a network issue occurs or the API returns invalid data, appropriate exceptions are caught, an error message is printed, and the function returns None. This ensures robust error handling and program stability when fetching external data. Meanwhile, the insertion_sort() function takes as input a list of players and returns that list sorted in descending order by the amount of points they have scored in a game.

The purpose of the *Question* class is to represent a trivia question, including its metadata and content. The class contains key attributes such as category, which stores the question's category, question_text, which holds the actual question, options, which is a list of possible answers, and correct_answer, which specifies the correct response. The from_api_response method is a static method that parses a JSON response from the OpenTrivia Database API and initializes a Question instance. It extracts the question data, combines the correct and incorrect answers into the options attribute, and randomizes their order to ensure that the correct answer is not always in the same position. The class interacts with the get_questions function, which fetches data from the API and relies on from_api_response to create and return a properly initialized Question object.

The purpose of the Player class is to represent a player in the trivia game by tracking their name and points earned throughout a game. The score is initialized at 0 and increased incrementally using the add_score method. While all correct answers currently earn 1 point, the option to vary the amount of points added is kept to ease future development. The subclasses ChildPlayer and AdultPlayer extend the Player class, each defining a default difficulty level (easy for children and medium for adults) through the class-level attribute DEFAULT_DIFFICULTY. The Player class interacts primarily with the TriviaGame class, which manages player instances, and the TriviaGameGUI class, where player scores are updated and displayed on the leaderboard.

The purpose of the TriviaGame class is to manage the state of the game and organize player data. Its key attributes include players, a list that stores all participating players, and round, which tracks the current round number during gameplay. Based on the specified player type (child or adult), the method add_player creates an instance of either the ChildPlayer or AdultPlayer subclass and appends it to the player's list. The TriviaGame class interacts with both the Player subclasses and the TriviaGameGUI class. It acts as the central data structure, maintaining player information and game progression, which is then displayed through the graphical interface.

The TriviaGameGUI class (not to be confused with the TriviaGame class) is responsible for managing the GUI. Its purpose is to provide an interactive interface for players, where questions, options, scores, and game progression are displayed.

The key attributes include root, which initializes the main GUI window, and round_player_index, which tracks the order of players within a round. Additional attributes such as question_label, options_frame, and leaderboard_text handle the display of questions, answer options, and real-time updates to the leaderboard in more detail. Five methods are implemented to manage the logical flow:

- **ask_question**: Fetches a question based on the current player's type (child/adult) using the get_questions() function and displays it.
- **submit_answer**: Processes the player's selected answer, updates their score if correct, and advances the game.
- **update_leaderboard**: Updates the leaderboard with current player scores and round numbers.
- **next_player**: Manages the turn order and increments the round counter.
- **end_game**: Ends the game, sorts and displays the final leaderboard.

To put all of these classes into work, the main program logic initializes and launches the trivia game by coordinating player setup and starting the GUI. First, it prompts the user to input the number of players, ensuring the input is a valid positive integer. It then collects each player’s name and type (child or adult), validating that the player type is either "child" or "adult" before adding the player to the game using the add_player method of the TriviaGame class. Once all players are added, the program initializes an instance of the TriviaGameGUI class, allowing the user to play the game.

4. Specifications

4.1. Technical specifications

As specified in the requirements, the entire source code is contained in a single file, “trivia.ipynb”. While a multi-module directory structure might have been more organized, this approach is slightly more portable and in line with the exam requirements. The program is designed to run on 3.11.9. To execute the program, the user will need the following libraries:

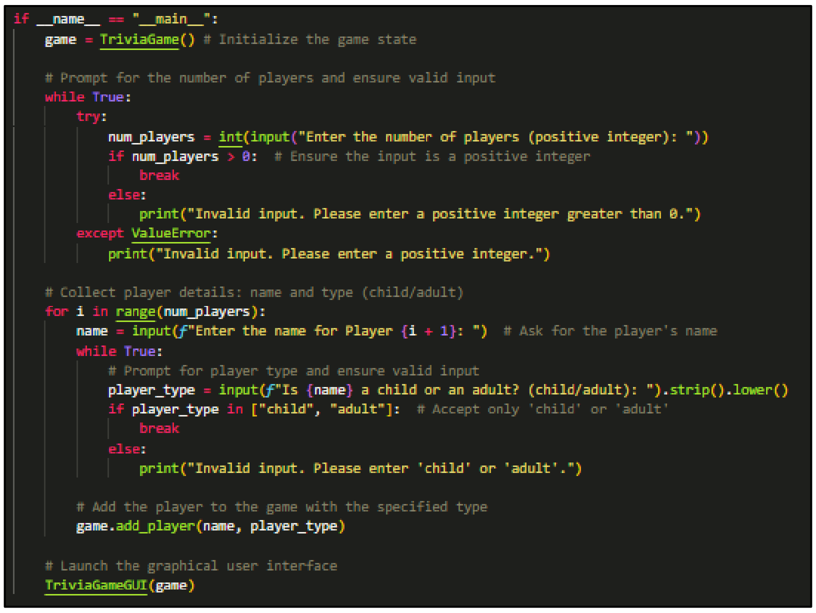
- **Requests**: A library for making HTTP requests to fetch trivia questions from the OpenTrivia Database API. This is *not* a Python standard library and will thus need to be *pre-installed.*
- **Random**: A standard library used to shuffle the options for each question.
- **Tkinter**: A standard library for creating the GUI.

The choice of these libraries ensures simplicity, portability, and ease of maintenance. The only dependency not part of the standard Python libraries, requests was deemed to be a strict necessity to fetch the questions and was thus included despite making the program slightly more cumbersome to execute. Although other third-party libraries might have provided a more aesthetically pleasing GUI; Tkinter was selected due to its availability in the Python standard library.

Since the program fetches questions dynamically, it requires a stable Wi-Fi connection. Should the connection fail or the API return invalid data, appropriate error-handling mechanisms are in place to inform the user, but the game would still be unplayable. The program can be executed on most operating systems that support Python 3, including Windows, macOS, and Linux.

4.2. Main program logic

To understand the workings of the program, two parts of the code are worth analyzing in further detail: The first is the main program logic (Figure 3).



```python
if __name__ == "__main__":
    game = TriviaGame() # Initialize the game state

    # Prompt for the number of players and ensure valid input
    while True:
        try:
            num_players = int(input("Enter the number of players (positive integer): "))
            if num_players > 0:  # Ensure the input is a positive integer
                break
            else:
                print("Invalid input. Please enter a positive integer greater than 0.")
        except ValueError:
            print("Invalid input. Please enter a positive integer.")

    # Collect player details: name and type (child/adult)
    for i in range(num_players):
        name = input(f"Enter the name for Player {i + 1}: ")  # Ask for the player's name
        while True:
            # Prompt for player type and ensure valid input
            player_type = input(f"Is {name} a child or an adult? (child/adult): ").strip().lower()
            if player_type in ["child", "adult"]:  # Accept only 'child' or 'adult'
                break
            else:
                print("Invalid input. Please enter 'child' or 'adult'.")

        # Add the player to the game with the specified type
        game.add_player(name, player_type)

    # Launch the graphical user interface
    TriviaGameGUI(game)
```

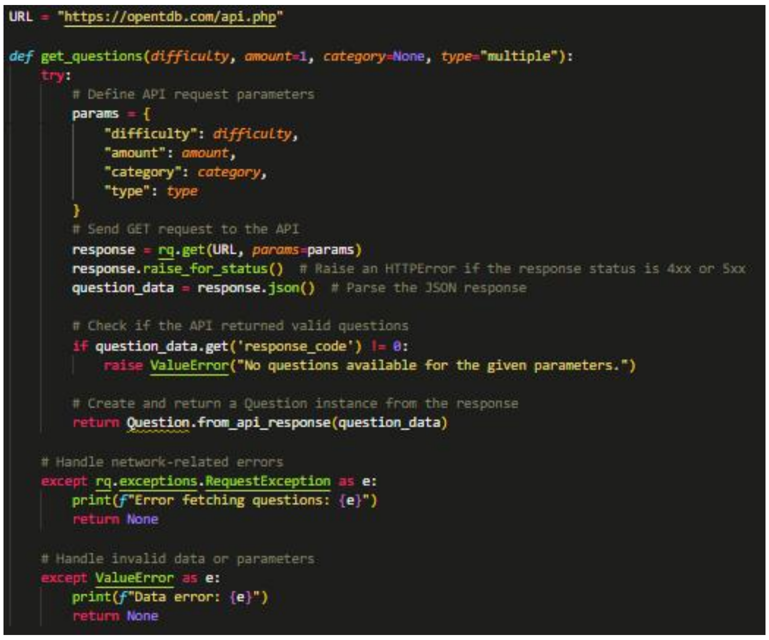
Figure 3: Main logic

The purpose of this section is to initialize the game by handling user inputs and launching the graphical user interface (GUI). The code begins with the standard Python entry point if __name__ == "__main__", to ensure that the code block executes only when the script is run directly.

The program then prompts the user for the number of players in a while loop. This design allows for repeated attempts if the user enters something other than a positive integer, which will be caught by the try/except logic. Although the current implementation accepts floats by truncating them to integers, this behavior has been deemed acceptable given the scope of the program. Following this, the program collects player details (name and type) for each player in a similar while loop. The player_type must be either "child" or "adult", and invalid inputs are rejected until a valid response is provided. Once validated, the player is added to the list of players within the TriviaGame instance using the add_player method. Finally, the program launches the GUI by creating an instance of the TriviaGameGUI class. This step connects the validated game data (players and initial state) to the interface.

4.3. Get_questions() function

The get_questions(), which fetches questions using the OpenTrivia API (figure 4) is also worth analyzing in more detail.



```python
URL = "https://opentdb.com/api.php"

def get_questions(difficulty, amount=1, category=None, type="multiple"):
    try:
        # Define API request parameters
        params = {
            "difficulty": difficulty,
            "amount": amount,
            "category": category,
            "type": type
        }
        # Send GET request to the API
        response = rq.get(URL, params=params)
        response.raise_for_status()  # Raise an HTTPError if the response status is 4xx or 5xx
        question_data = response.json()  # Parse the JSON response

        # Check if the API returned valid questions
        if question_data.get('response_code') != 0:
            raise ValueError("No questions available for the given parameters.")

        # Create and return a Question instance from the response
        return Question.from_api_response(question_data)

    # Handle network-related errors
    except rq.exceptions.RequestException as e:
        print(f"Error fetching questions: {e}")
        return None

    # Handle invalid data or parameters
    except ValueError as e:
        print(f"Data error: {e}")
        return None
```

Figure 4: get_questions() function

The get_questions function is responsible for retrieving trivia questions from the OpenTrivia Database API. It begins by using the global variable URL, which stores the base API endpoint. The function includes the parameters difficulty, amount, category, and type. While the difficulty is a flexible input controlled by the program to differentiate between children and adults, the amount, category, and type parameters are standardized. This design simplifies the program's current functionality but leaves room for future development where these options could be made configurable.

The function converts the API response into a JSON format using response.json(), to turn it into a structured dictionary. The parsed data is validated to ensure the API returns valid questions. If the response_code indicates failure, a ValueError is raised to signal invalid query parameters. Network-related issues, such as connectivity problems or server errors, are caught using a try/except block that handles RequestException. Data-related errors, such as invalid API responses, are addressed with a separate ValueError block. In both cases, informative error messages are printed, and the function returns None to signal failure. Assuming no errors occur, the function processes the validated JSON data and returns a Question instance by calling the from_api_response method.

5. Results & discussion

When running the app, it can be seen that the program fulfills the functional requirements through the simple GUI (figure 5). It fetches a question of a random category as intended, with an adjustment in difficulty based on the player type. It presents the questions clearly, along with the name of the player intended to answer. It correctly tracks the leaderboard on the left and updates immediately when a question is answered, after providing direct user feedback on correctness (Appendix 8.1.). A minor shortcoming is that the GUI has trouble handling special characters, but not to a point where the questions are incomprehensible (Appendix 8.2.). Thus, the program does indeed provide a playable trivia game as intended.

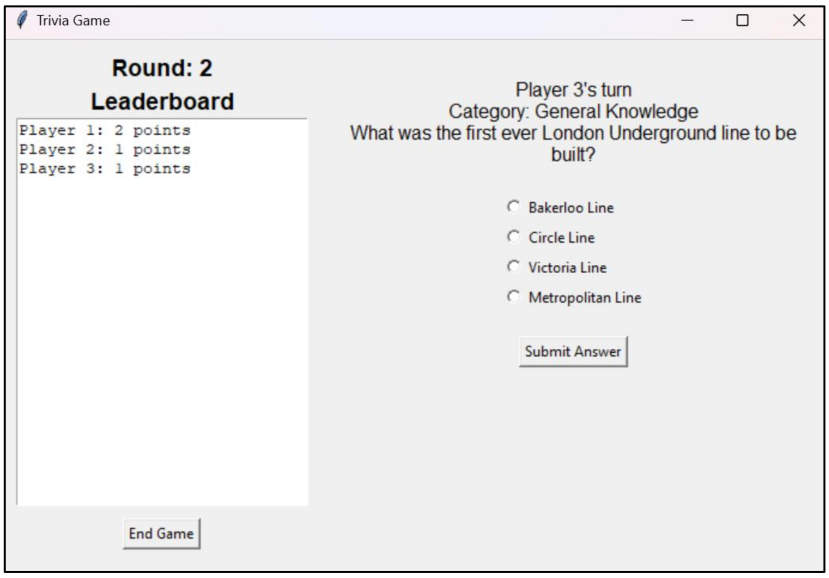


Figure 5: GUI

With regards to the non-functional requirements, the code structure provides a flexible, maintainable, and portable program as alluded to in previous sections. In addition, it is rather easy to use, ensuring that it is also playable for non-technical audiences (at least after the GUI is launched). It is deemed that the game is also "replayable" since the same question was not seen twice during all the testing done during development. Two minor shortcomings are noticed. Firstly, the "easy" questions are arguably still of difficulty beyond what is expected knowledge of a young child. This could be improved to make the game even more family-friendly. Secondly, it would be more user-friendly if the player count and information prompts were handled in the GUI instead of the coding environment. The game has been tested on multiple computers without encountering any problems related to compatibility or performance, providing tentative evidence for the meeting of these requirements. However, more extensive user testing would be ideal to ensure that no fringe cases are left uncovered.

During implementation, a surprising amount of resources was spent hedging against wrong input by the user in the player count and information prompts, and the GUI. To address the user prompt issues, a thoroughly designed try/except flow was implemented in the main logic, designed to ensure that the user is providing the right input, and gets to try again if they don't. In addition, the GUI was very time- and code-consuming to design. To prevent the resource investment in front-end design from getting out of hand, the design was kept to a bare functional minimum, with the optionality to improve it further. The solutions to these challenges, as well as the fulfillment of the non-functional requirements, all rest on the principle of balancing between current functionality and options for future development. This taught the developers a lot about reaching a pragmatic solution that fulfills the functional requirements while being sufficiently modular and flexible to be easily improved.

6. Conclusions and future work

In conclusion, the functional and non-functional requirements, as well as the constraints, are met to a satisfactory degree, and the code provides a useable and engaging trivia game. There are a few shortcomings to be addressed in future development, namely the handling of special characters, the handling of floats in the user count prompt, the difficulty of children's questions, and moving the input of player count and information to the GUI. These should be addressed in the next iteration of development.

In addition to these low-hanging fruits, some possible improvements could be addressed in the long term. The first one is to make the gameplay more configurable, e.g. by allowing players to choose categories and type (multiple choice/Boolean), and make a more sophisticated scoring mechanism based on this. In addition, there is room for improvement in the aesthetics of the GUI, which are currently far from cutting edge, despite fulfilling the functional requirements. This could be made using more sophisticated libraries such as PyQt, Kivy, or customtkinter. However, this should be weighed against the additional dependencies that this would introduce. Another potential improvement with a more substantial resource investment would be to introduce more game modes, such as answer timers, a playing board, or similar. This would bring more variety to the game beyond the question contents, which would enhance the "replayability".

7. Evaluation of learning objectives

7.1. Understanding of fundamental Python concepts

The program makes use of variables, loops, functions, classes, and "if" statements. For example, while loops are used to validate user input for the number of players and their types, while "if statements" are used to make a simple control flow based on the user's input. Throughout the code, the appropriate choices are made in what techniques to use, and it is thus the proprietary assessment by the author that this learning objective is *fulfilled.*

7.2. Design and implementation using appropriate syntax and features

The program adheres to Python conventions and best practices, including clear function and class definitions, meaningful docstrings, and structured error handling. The program runs without syntax errors, and the logic is separated into functions, classes, and the main program flow. It is thus the assessment of the author that this learning objective is also *fulfilled*.

7.3. Use of imperative, declarative, and object-oriented features

The program correctly utilizes OOP principles, including encapsulation, inheritance, and polymorphism. Inheritance is demonstrated by the Player class, which serves as a base class for the ChildPlayer and AdultPlayer subclasses extending its functionality and defining a DEFAULT_DIFFICULTY attribute. Imperative programming structures are used for input validation and control flow, while declarative features are leveraged to interact with the external API. Thus, the author assesses that this objective is also *fulfilled*.

7.4. External libraries and APIs

The program integrates the third-party requests library to interact with the OpenTrivia Database API for retrieving trivia questions. The get_questions function constructs and validates API requests, ensuring the program dynamically fetches data in real-time. This learning objective is thus also *fulfilled*.

7.5. Algorithm analysis and program complexity

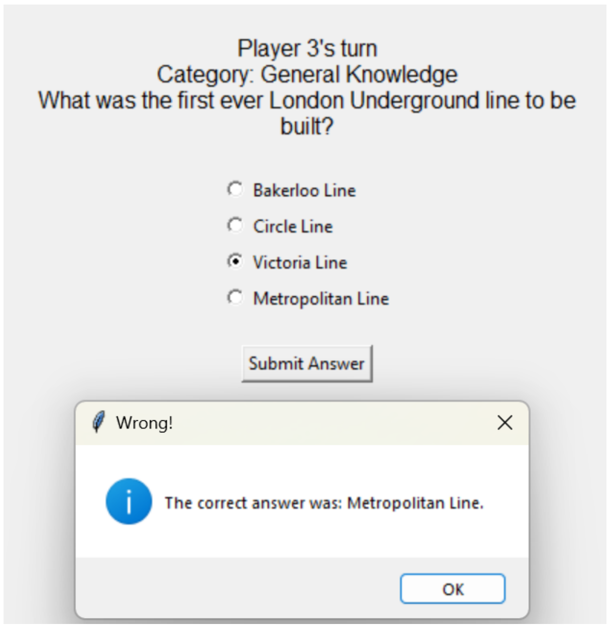
A deliberate effort has been made to account for the algorithmic complexity of the program, as shown by the analysis of sorting algorithms in section 3.1. The learnings from the choice are used to choose the sorting algorithm that balances performance with complexity and ease of implementation given the specific conditions of this program. Thus, this objective is *fulfilled*.

7.6. Critical awareness of methodological choices and written skills

Throughout this report, sections 3-6 in particular, the methodological choices are continuously assessed critically. A great effort has been made to reflect on the advantages and disadvantages of each decision, and the areas for improvement are identified analytically and objectively. Thus, it is the proprietary assessment that this objective is also *fulfilled*, meaning that all six learning objectives are met.

8. Appendix

8.1. Evaluation of answer correctness



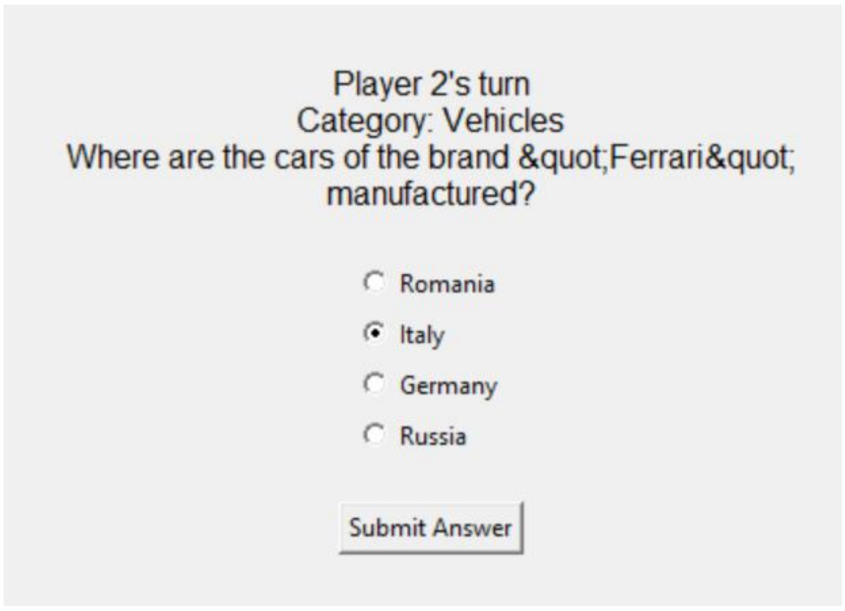
Player 3's turn
Category: General Knowledge
What was the first ever London Underground line to be built?

- ○ Bakerloo Line
- ○ Circle Line
- [✓ Victoria Line]
- ○ Metropolitan Line

Wrong!

The correct answer was: Metropolitan Line.

8.2. Handling of special characters



Player 2's turn
Category: Vehicles
Where are the cars of the brand &quot;Ferrari&quot; manufactured?

- ○ Romania
- [✓ Italy]
- ○ Germany
- ○ Russia