Python-Quiz-for-Summative-AE2

Click here to get instructions as to how to install the quiz application

Introdcution

This is a databricks quiz and is a minimal viable product (MVP) developed for new users of databricks to test their knowledge around data modelling, the medallion architecture and other data analyst themed questions tailored to power BI developers. Part of work within the data side of Department for Education has been migrating SQL code and data within warehouse server databases to a much more modern online data store facility such as databricks that stores loud-based data within a Lakehouse architecture. This quiz has been developed to help beginner data modelers and power BI developers bridge this transition and also highlight area of improvement and knowledge gaps they might have.

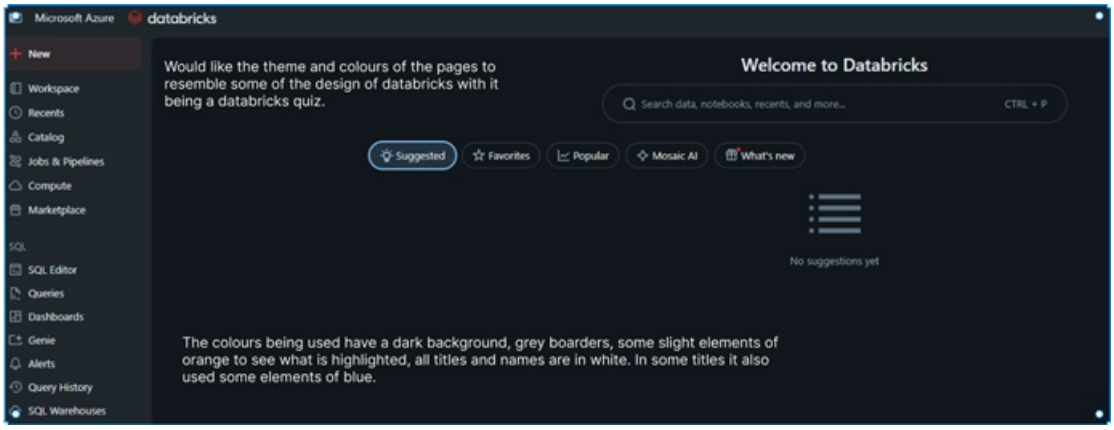
The databricks quiz is a Python and Tkinter based application that consists of 15 multiple choice questions and also collects the usernames of those who have completed the quiz and the score they got. This data is stored on a local CSV file in the format of “Name”, “Score”, “Total”, “Percent”, and “Timestamp”.

These bits of information are useful to indicate which users might need some more support or time focused to learning and development to help with this transition. The quiz itself should take no longer than 15mins to complete and provide a simple solutions to help managers have a better understanding of their team’s confidence with databricks and data modelling.

Design

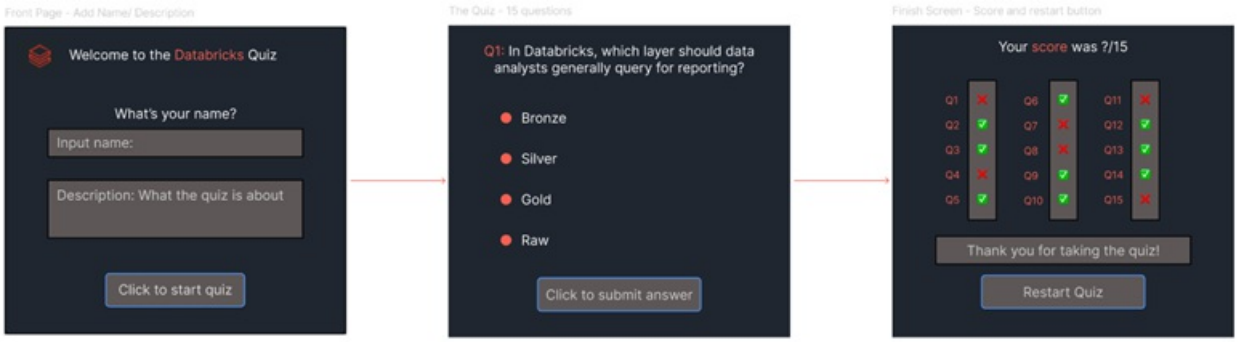
The layout of the quiz and user journey includes 3 main screens: - The start screen is where the user will input their name which their score will be stored against at the end, a basic description of the theme of the quiz, and a button at the bottom which will start the quiz and go to the next screen. - The main quiz screen, this will consist of the 15 multiple choice questions with 4 options for each question to choose from. Each answer will provide a pop up that says if you got the questions correct or incorrect as well as an explanation of the answer. There is also a submit and next question button at the bottom of this page too. - Once all questions have been answered the ending screen will appear and give you your total score and a breakdown of each questions results as well as another button if the user wishes to take the quiz again (restart).

I wanted the style of the quiz to also match the look of databricks.



This is an image of what the main home screen looks like as well as my comments relating to the colour palette which will be the main way for users to have that familiarity with the quiz and databricks. I then pulled each colour into a colour palette and designed the wireframe of the quiz application.

The image below represent a baseline for the visual appearance of the quiz and well as showcasing the navigational flow for users and planned screen layouts matching the colour palette shown above.



Functional Requirements

Functional Requirements

ID	Requirement
FR1	The application must allow the user to input and store their name.
FR2	The application must allow navigation between questions.
FR3	The application must allow the user to submit their selected answer for each question.
FR4	The application must calculate and display a score out of 15.
FR5	The application must store results in a CSV file associated with the username.
FR6	The application must enforce a minimum aspect ratio to prevent visual distortion.
FR7	The application must include a restart button.

Non Functional Requirements

Non-functional Requirements

ID	Requirement
NFR1	The application must use the following colour scheme: BG_DARK = "#1F2630", WHITE = "#E2E2E2", ACCENT = "#F86153", GREY = "#5E5757", BORDER = "#000000", BLUE = "#4D91EA".
NFR2	The font size must be accessible and readable for all users.
NFR3	The application must display a message box showing "correct" or "incorrect".
NFR4	The application must show an answer explanation after each question.
NFR5	The application must display a message indicating an answer must be selected before proceeding.
NFR6	The application must include name-input error handling.

Tech Stack Outline

```
[Python 3](https://docs.python.org/3/) – core programming language
```

```
[Tkinter](https://docs.python.org/3/library/tkinter.html) – desktop graphical user interface
```

```
[csv](https://docs.python.org/3/library/csv.html) – local data storage in CSV format
```

```
[re](https://docs.python.org/3/library/re.html) – regular expressions for input validation
```

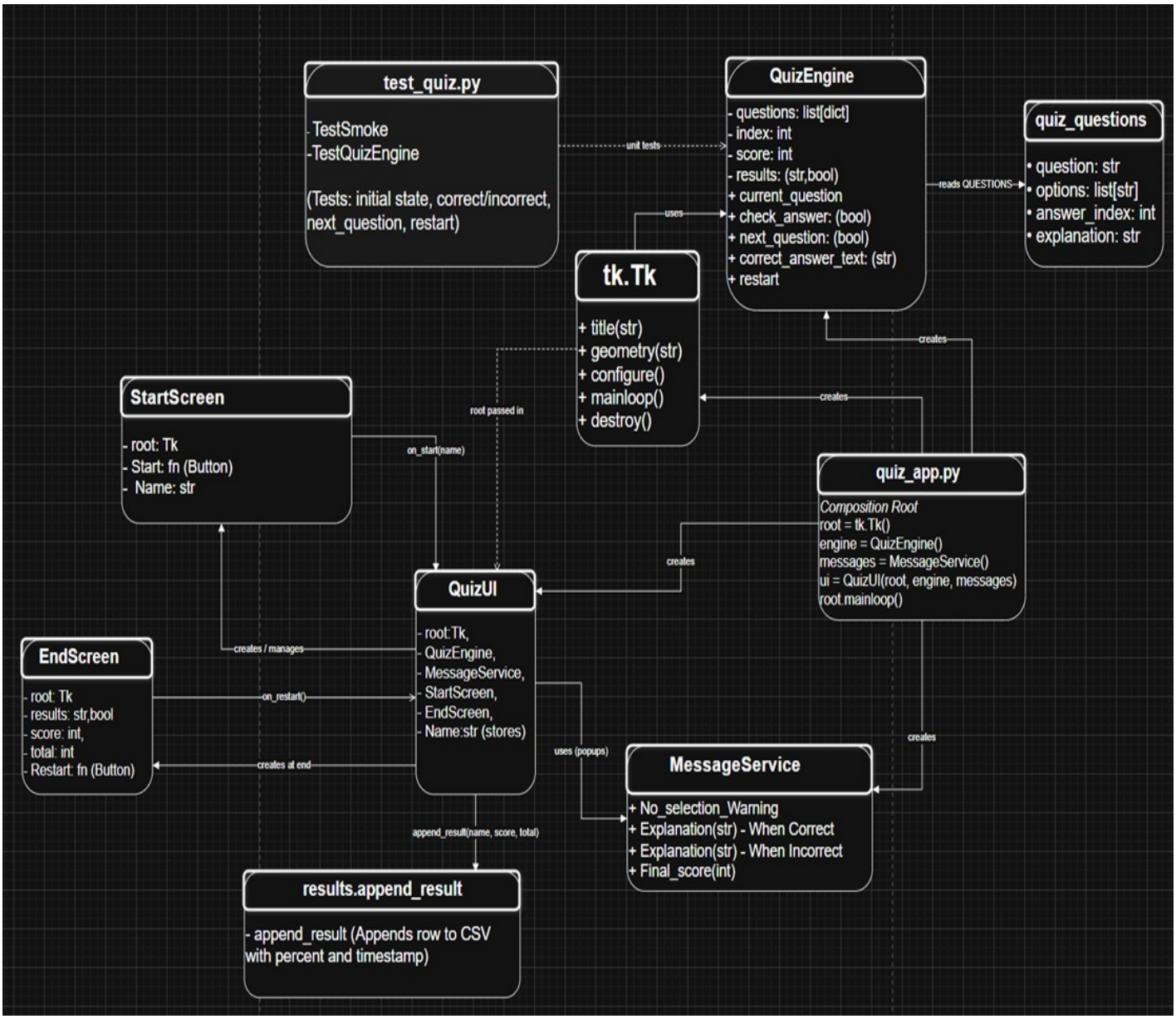
```
[datetime](https://docs.python.org/3/library/datetime.html) – timestamp generation
```

```
[unittest](https://docs.python.org/3/library/unittest.html) – automated unit testing
```

```
[Path](https://docs.python.org/3/library/pathlib.html)  – where the results will be store
```

```
[Message box](https://docs.python.org/3/library/tkinter.messagebox.html) – message
```

Code Design



Here is the code design I made in draw.io. This showcases the 9 different files that will be present within the code. They consist of:

- Quiz Engine – Code that manages everything that isn't the user interface (UI) such as keeping track of the score, pulling the questions, question index, question explanations and checking if the selected answer is correct.
- Quiz App – Links all the UI elements and Quiz Engine together, this is also where you run the main quiz application. (run this bit of code and it will run the entire quiz)

All the Tkinter graphical code:

- Quiz UI – This is the main graphical interface and is responsible for overall design of each page, it also creates the start and end pages to switch between the three different views. It interacts with the quiz engine through the quiz app as well as any pop-up messages.
- Start screen – All the graphics for the start screen, also including some error handling for the username.

- Ending screen – Graphics for ending page that showcases the users score and if they would like to restart the quiz.

Extras

- Messages – All the pop-up messages for correct or incorrect answers including errors.
- Quiz Questions – All the questions, options, answers and explanations.
- Test Quiz – This includes all the unittest scripts that specifically targets the quiz engine testing for bugs/ errors in the scoring, switching question, incorrect/ correct answers etc.
- Tk.Tk – This provides all the support the for Tkinter module. Code that uses or is connected to Tkinter must call on this through root.tk to become the main application window and is connected to Quiz UI, Start screen, End screen, and the quiz engine.

Development, Testing and Coding Documentation

First Developing the quiz engine

The QuizEngine is an object-oriented Python class that acts as the core logic layer for the quiz application. The main responcibilities of this code is managing the current quiz state by: tracking user progress, verifying answers, and preparing data for the UI layer. This is utalising OOP principles such as encapsulation to build the fields and user attributes, state management to hold and update these attribute values such as score, and method-driven interactions so that things such as score are only updated when check_answer runs.

The constructor (**init**) automatically runs when creating a new QuizEngine object. It sets up the quiz's internal state, including the question list, index tracking, score counter, and answer history.

```
class QuizEngine:
    def __init__(self):  # This means that it runs automatically when a new QuizEngine() object is made.
        self.questions = QUESTIONS
        self.index = 0
        self.score = 0
        self.results = []  # Keep Track of what question a user got right or wrong for results at the end.
```

Answer Checking

This method verifies if the user's choice is correct, updates the score, and records the result.

```
def check_answer(self, choice_index: int) -> bool:
    question = self.current_question()
    correct_index = question["answer_index"]
    is_correct = (choice_index == correct_index)

    self.results.append((question["question"], is_correct))

    if is_correct:
        self.score += 1
    return is_correct
```

This section of the code pulls from another file called quiz questions. Quiz Questions is layed out in a way where it supplies the question, options, answer in the form of an answer_index as well as an explaination which contains the reasoning behind the answers. It is formatted like the example below.

```
QUESTIONS = [
{
"question": "Q1: In Databricks, which layer should data analysts generally query for reporting?",
"options": ["Bronze", "Silver", "Gold", "Raw"],
        "answer_index": 2,
        "explanation": "Gold contains curated, business-ready tables (facts/dimensions) designed for analytics and BI consumption."
},
```

Moving to the next question

This controls the progression of the quiz, it does this by utalising the quiz index to check if there is another question. This also helps as part of the UI will want to know if there is any more questions, if the boolian is set to false it will move to the ending screen.

```
    def next_question(self) -> bool:

    # Moves to the next question.
    # Returns True if another question exists, otherwise False.

        self.index += 1
        return self.index < len(self.questions)
```

Restart option

An option to restart the quiz, setting the index, score to 0 and clearing all the results. (Replay feature without having to close the app)

```
    def restart(self):
        self.index = 0
        self.score = 0
        self.results = [] # Restart the quiz and set th quiz index and user score to 0 again
```

Engine Testing using Test Driven Development (TDD)

While building the engine, at the same time I also built the testing script so that any continusous intergation(CI) within the engine was also matched with test driven development (TDD). Test Driven development is used here to make sure that the porject itself behaves as it is supposed to throughout development, this mean that I can make changes to the engine and the questions without the worry of regression (when one fix breaks another element accidently). The test is ran manually from within visual studio code, 7 tests are ran in total each time and they include: Smoke Test, Question test, Quiz Engine test, **init** test, incorrect test, correct answer input and finally restart test. Example of the code:

Smoke Test

```
class TestSmoke(unittest.TestCase):

    def test_unittest_runs(self):
        self.assertTrue(True) # True is always true so should always pass

    def test_questions_load(self):
        self.assertGreater(len(QUESTIONS), 0) # Make sure that questions is imported correctly, file isn't empty or broken
```

For testing I used a module called unittest, the smoke test is designed to make sure unittest framework itself is working before running any real tests. This helps identify any environmental errors such as problems with running python within visual studio code or any broken file paths. Here the basic tests include: Testing that True is True (which should always pass) as well as if the questions are being imported correctly which creates a reliable starting point for the test driven development.

Testing quiz engine

```
class TestQuizEngine(unittest.TestCase): # Testing the quiz engine
    def setUp(self):
        self.engine = QuizEngine() # Creates fresh engine, dosen't interfere with other tests

    def test_initial_state(self):
        self.assertEqual(self.engine.index, 0)
        self.assertEqual(self.engine.score, 0)
        self.assertEqual(self.engine.results, [])
```

This sets up a new isolated quiz engine instance and tests that the inital state of the quiz must always be index=0, score=0, and results=[]. It makes sure that the default state of the quiz always remains the same thus also testing regressions that would change this state such as the restart button not reseting the quiz value attributes.

Testing correct answers

```
def test_correct_answer_increments_score(self):
    q = self.engine.current_question()
    correct = q["answer_index"]

    result = self.engine.check_answer(correct)

    self.assertTrue(result)
    self.assertEqual(self.engine.score, 1)
    self.assertEqual(len(self.engine.results), 1)
```

Testing incorrect answers

```
def test_incorrect_answer_does_not_increment_score(self):
    q = self.engine.current_question()
    wrong = (q["answer_index"] + 1) % len(q["options"])

    result = self.engine.check_answer(wrong)

    self.assertFalse(result)
    self.assertEqual(self.engine.score, 0)
    self.assertEqual(len(self.engine.results), 1)
```

This tests both sides of the user input, when either a correct answer or incorrect answer is chosen that the score is reflected in the way it should be. As part of the test driven development, it ensures that the answer_index is working as it should do. That the quiz_engine and quiz_questions (where the answer_index is stored) are working in tandem with one-another. By testing for both the correct and incorrect user paths it makes sure that testing is more robust, that either side-effect is consistant with the method's declared semantics.

Testing Restart

```
def test_restart_resets(self):
    self.engine.check_answer(self.engine.current_question()["answer_index"])
    self.engine.next_question()
    self.engine.restart()

    self.assertEqual(self.engine.index, 0)
    self.assertEqual(self.engine.score, 0)
    self.assertEqual(self.engine.results, [])
```

Finally, it tests that upon restart the index, score, and results have all reset to the default value. This tests that repeatable sessions are not only possible but work as intended. That when the questions index finish, the engine picks up that it has transitioned set the next_question bool to false (correct terminal behaviour).

The output of the test looks like this when ran successfully

```
 python -m unittest -v test_quiz.py
test_correct_answer_increments_score (test_quiz.TestQuizEngine.test_correct_answer_increments_score) ... ok
test_incorrect_answer_does_not_increment_score (test_quiz.TestQuizEngine.test_incorrect_answer_does_not_increment_score) ... ok
test_initial_state (test_quiz.TestQuizEngine.test_initial_state) ... ok
test_next_question_works (test_quiz.TestQuizEngine.test_next_question_works) ... ok
test_restart_resets (test_quiz.TestQuizEngine.test_restart_resets) ... ok
test_questions_load (test_quiz.TestSmoke.test_questions_load) ... ok
test_unittest_runs (test_quiz.TestSmoke.test_unittest_runs) ... ok

----------------------------------------------------------------------
Ran 7 tests in 0.004s

OK
```

And this when errors occour highlighting exactly which lines within Quiz_Engine have caused errors within the unittest.

```
 python -m unittest -v test_quiz.py
test_correct_answer_increments_score (test_quiz.TestQuizEngine.test_correct_answer_increments_score) ... ERROR
test_incorrect_answer_does_not_increment_score (test_quiz.TestQuizEngine.test_incorrect_answer_does_not_increment_score) ... ERROR
test_initial_state (test_quiz.TestQuizEngine.test_initial_state) ... ok
test_next_question_works (test_quiz.TestQuizEngine.test_next_question_works) ... ERROR
test_restart_resets (test_quiz.TestQuizEngine.test_restart_resets) ... ERROR
test_questions_load (test_quiz.TestSmoke.test_questions_load) ... ok
test_unittest_runs (test_quiz.TestSmoke.test_unittest_runs) ... ok

======================================================================
ERROR: test_restart_resets (test_quiz.TestQuizEngine.test_restart_resets)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\(Directory Hidden for Security)\test_quiz.py", line 72, in test_restart_resets
    self.engine.check_answer(self.engine.current_question()["answer_index"])
                             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\(Directory Hidden for Security)\quiz_engine.py", line 13, in current_question
    return self.questions[self.index] # Return the question dictionary for the current index.
           ^^^^^^^^^^^^^^
AttributeError: 'QuizEngine' object has no attribute 'questions'. Did you mean: 'quesions'?

----------------------------------------------------------------------
Ran 7 tests in 0.007s
FAILED (errors=4)
```

Quiz Graphical User Interface

This is split up into three main sections, the start_screen, quiz_ui (question screen) and the end_screen. The Quiz_ui is the bridge between the user interface and the quiz logic that is all stored as part of the quiz_engine. It is responsible for displaying questions, handling user selections, validating submissions, managing quiz progression, and triggering the final results screen. The class uses internal state management, event-driven methods, controlled flow transitions, and data passing between components to keep the quiz consistent and responsive. It also ensures user input is handled safely, explanations are delivered correctly, and results are recorded reliably at the end of the quiz session.

The code mainly revolves around event-driven programming, the idea that the user triggers actions within the UI and the app responds accordingly. (e.g. clicking submit or next)

```
    # submit or next, whiching from one button to the other when a question in answered or submitted
    def _on_submit_or_next(self):
        if self.awaiting_submit:
            self._submit_answer()
        else:
            self._next_step()

    def _submit_answer(self):
        choice = self.selected.get()
        if choice == -1:
            self.messages.warn_no_selection() # If no answer is selected then it outputs the didn't choose anything warning message
            return

        q = self.engine.current_question()
        explanation = q.get(
            "explanation",
            "This answer aligns with Databricks & Medallion best practices." # This gives a failsafe, if no explainataion is given t
        )
        is_correct = self.engine.check_answer(choice) # This updates the score and results list
```

This is one such example where I have used event-driven methods by using method switching between the submit button and the next question button keeping input handling predictable and centralised as the user can only click one button at any one time. It avoids the potential errors that come with duplicated logic and ensures that the flow of the application always moves to the intended controlled checkpoint.

This is also a good example of error handling where the UI uses the module Message box to safely warn the users when their actions are invalid such as in this case where "warning no selection" will appear if the user clicks of submit answer without having chosen an answer option. It can also be useful to prevent other elements of code breakage such as the results table not working as it would be missing a question input.

Switching Screens

```
if self.engine.next_question():
    self.show_question()
else:
    append_result(self.user_name, self.engine.score, len(self.engine.questions))
```

This is a clean example of state transition logic, moving from quiz mode → end screen based on engine state.

Graphical Elements

```
        # Build new Radiobuttons that match the figma colour scheme
        for i, opt in enumerate(q["options"]):
            rb = tk.Radiobutton(
                self.options_frame,
                text=opt,
                variable=self.selected,
                value=i,
                font=("Arial", 14),
                fg=WHITE,
                bg=BG_DARK,
                activebackground=BG_DARK,
                activeforeground=WHITE,
                selectcolor=BG_DARK,     # keeps indicator background matching the dark bg
                anchor="w",
                justify="left",
                wraplength=720
            )
            rb.pack(anchor="w", fill="x", pady=8)
            self.option_buttons.append(rb)
```

A fair amount of the rest of the quiz_ui code looks like this, filled with different visual parameters that have been taylored to match the databricks example and colour palette. Visual parameters (fg/bg/active*/selectcolor/anchor/justify/wraplength) make options readable, themed, and neatly aligned in a dark UI. The actual radio button loop itself builds one button per option binding them to a shared IntVar for mutually exclusive selection.

Start screen

The code here is almost identical to that of the quiz UI in terms of the visual elements however there are also some noticeble changes that are worth mentioning. Such as the more in-depth error handeling within the start screen to restrict the user to only input a valid name, this input validation is monitored via re or Regular expression operations.

```
 # Letters and hyphens only, 3–15 chars, must start/end with a letter
NAME_REGEX = re.compile(r"^[A-Za-z](?:[A-Za-z\- ]{1,20})[A-Za-z]$")
```

```
    # HANDLER WITH VALIDATION
    def _handle_start(self):
        raw = (self.name_entry.get() or "").strip()

        # Treat placeholder as empty
        if raw == "" or raw == "Input name:":
            messagebox.showwarning(
                "Name required",
                "Please enter your name (3–25 letters, hyphens allowed)."
            )
            self.name_entry.focus_set()
            self.name_entry.select_range(0, tk.END)
            return

        # Validate with regex: letters + hyphens, 3–15 chars, must start/end with a letter
        if not NAME_REGEX.match(raw):
            messagebox.showwarning(
                "Invalid name",
                "Your name must be 3–15 characters, contain only letters and hyphens (-), "
                "and begin and end with a letter.\n\nExamples:\n• Elliot\n• Pacey-Carrier"
            )
            self.name_entry.focus_set()
            self.name_entry.select_range(0, tk.END)
            return

        # If valid, proceed
        name = raw
        self.hide()
        self.on_start(name)
```

Summary of start screen input validation/ error handling code

The _handle_start method validates the name entered on the start screen and only proceeds if it's acceptable. It first normalizes the input, then blocks empty/placeholder values and rejects strings that don't match a regex rule (letters and hyphens, 3–15 chars, must start/end with a letter). On failure, it shows a warning dialog, restores keyboard focus to the field, and pre-selects the text for easy correction. On success, it hides the start screen and invokes on_start(name) to continue the app flow. The structure uses early returns, messagebox-based feedback, and focus management to provide robust, user-friendly error handling.

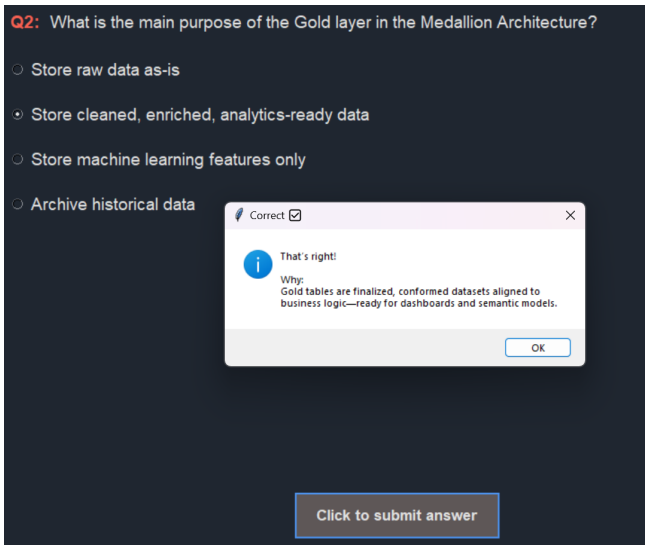
Results table

Finally, there is the results table, this is connected to the main quiz UI and stores the "name", "score", "total", "percent" and "timestamp" of all the users who have done the quiz as well as the time they did it. This is really useful as it stores user progress as part of a CSV file and due to the ability to re-run the quiz it could also showcase progress if the user wanted to take the quiz again after some databricks training.

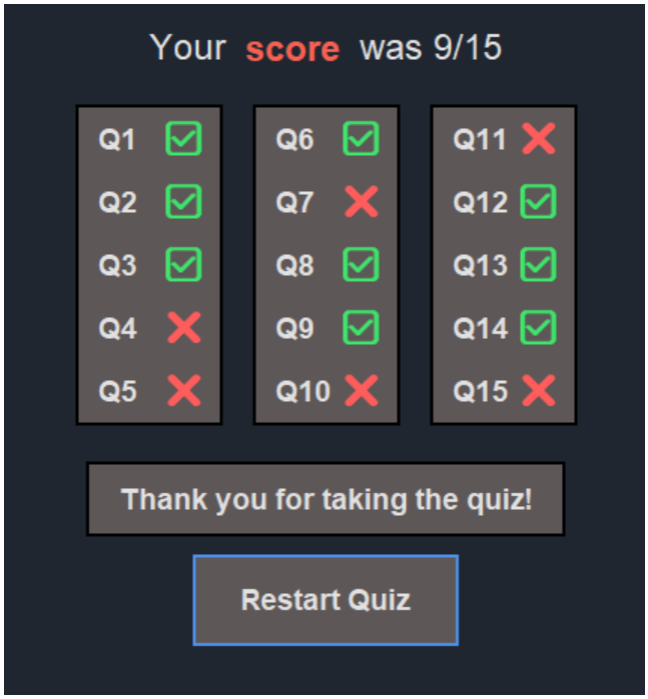
```
 RESULTS_CSV = Path("quiz_results.csv")
def append_result(name: str, score: int, total: int, csv_path: Path | str = RESULTS_CSV) -> None:
    csv_path = Path(csv_path)
    file_exists = csv_path.exists()
    # Local time with timezone info
    ts = datetime.now().astimezone().isoformat(timespec="seconds")
    percent = round((score / total) * 100, 2) if total > 0 else 0.0
    # Ensure parent folder exists
    csv_path.parent.mkdir(parents=True, exist_ok=True)
    with csv_path.open(mode="a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        if not file_exists:
            writer.writerow(["name", "score", "total", "percent", "timestamp_iso"])
        writer.writerow([name, score, total, percent, ts])
```

Evaluation

Here is a visual of the end product,



(cropped slighty but button in central in the app)



Overall, I was really happy with the end result of the quiz. Development for the most part went quite smoothly and I was pleased with the quiz engine and overall design matching very closely to the initial Figma GUI prototype. This however also but a lot of time constraints on the project and in future I would probably prefer to experiment with more modern GUI python modules. I went for quiz a modern design using a slightly outdated Tkinter module which meant the UI coding felt quite clunky at times, that being said I would use Tkinter again to create application prototypes as I think it is fairly straight forward to utilise.

The quiz itself had positive feedback, users could quite easily run the quiz application through their own machine however this did take a bit of setting up at first. In the future I would like to make this more accessible, as everyone who attempted the quiz was already quiz familiar with visual studio code so those that might not would have more of a difficult time. Users also mentioned that while the quiz was a good indicator for beginners using databricks, they mentioned that future iterations should include harder questions to not only highlight those that need training but staff that also already have a significant background with databricks.