Python Code

```python
import tkinter as tk
from quiz_engine import QuizEngine
from quiz_ui import QuizUI
from messages import MessageService

root = tk.Tk()
engine = QuizEngine()
messages = MessageService()
ui = QuizUI(root, engine, messages)
root.mainloop()

from quiz_questions import QUESTIONS # Importing the questions from quiz_questions

# Used from the iteratve assignment as basis for quiz

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.

    def current_question(self):
        return self.questions[self.index] # Return the question dictionary for the current index.

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

    # Checks whether the chosen answer is correct.
    # Records the result for the end screen.
    # Returns True if correct, False if not.

        self.results.append((question["question"], is_correct)) # Stores a tuple: the question text and whether the user got it right.

        if is_correct:
            self.score += 1
        return is_correct # Update score if answer input is correct or not.

    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)

    def correct_answer_text(self) -> str:
        q = self.current_question()
        return q["options"][q["answer_index"]] # This is important in helping the QUIZ_UI regognise that an answer selected in incorrect and to display the incorrect message.

    # If the boolean would return false e.g. no more questions then it will move the the ending_screen.

    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
# end_screen.py
import tkinter as tk

# Palette (same as start/quiz UI)
BG_DARK    = "#1F2630"
FG_TEXT    = "#E2E2E2"
ACCENT     = "#F86153"
GREY       = "#5E5757"
BORDER     = "#000000"
BLUE       = "#4D91EA"

CHECK = "✅"
CROSS = "❌"


class EndScreen:

    def __init__(self, root, results, score, total, on_restart):
        self.root = root
        self.results = results
        self.score = score
        self.total = total
        self.on_restart = on_restart

        # MAIN
        self.frame = tk.Frame(root, bg=BG_DARK)
        self.frame.pack(fill="both", expand=True)

        # HEADER: "Your score was X/15"
        header = tk.Frame(self.frame, bg=BG_DARK)
        header.pack(pady=(20, 12))

        tk.Label(
            header, text="Your ", font=("Arial", 18), fg=FG_TEXT, bg=BG_DARK
        ).pack(side="left")

        tk.Label(
            header, text="score", font=("Arial", 18, "bold"), fg=ACCENT, bg=BG_DARK
        ).pack(side="left")

        tk.Label(
            header, text=f" was {self.score}/{self.total}", font=("Arial", 18),
fg=FG_TEXT, bg=BG_DARK
        ).pack(side="left")

        # RESULTS GRID (3 columns x 5 rows)
        # Prepare data as boolean list by question index
        # If results length < total (edge cases), pad with False
        flags = [is_correct for (_q, is_correct) in self.results]
        if len(flags) < self.total:
            flags += [False] * (self.total - len(flags))
        flags = flags[:self.total]

        grid = tk.Frame(self.frame, bg=BG_DARK)
        grid.pack(pady=(8, 16))

        # Helper to build one grey column with black border
        def make_column(parent):
            outer = tk.Frame(parent, bg=BORDER)
            outer.pack(side="left", padx=10)
            inner = tk.Frame(outer, bg=GREY)
            inner.pack(padx=2, pady=2)
            return inner

        col1 = make_column(grid)
        col2 = make_column(grid)
        col3 = make_column(grid)

        # Place Q1..Q15 across the three columns
        # Column 1: 1..5, Column 2: 6..10, Column 3: 11..15
        def add_rows(col_frame, start_q_idx, end_q_idx):
            for i in range(start_q_idx, end_q_idx + 1):
                # i is 1-based question number; flags index is i-1
                ok = flags[i - 1] if (0 <= i - 1 < len(flags)) else False
                symbol = CHECK if ok else CROSS
                fg_symbol = "#3EE46B" if ok else "#FF5C5C"

                row = tk.Frame(col_frame, bg=GREY)
                row.pack(anchor="w", padx=10, pady=6)

                tk.Label(
                    row, text=f"Q{i}", font=("Arial", 14, "bold"),
                    fg=FG_TEXT, bg=GREY, width=3, anchor="w"
                ).pack(side="left")

                tk.Label(
                    row, text=symbol, font=("Arial", 16, "bold"),
                    fg=fg_symbol, bg=GREY, width=2, anchor="w"
                ).pack(side="left")

        add_rows(col1, 1, 5)
        add_rows(col2, 6, 10)
        add_rows(col3, 11, 15)

        # THANK YOU STRIP
        thanks_border = tk.Frame(self.frame, bg=BORDER)
        thanks_border.pack(pady=(8, 12))

        thanks = tk.Label(
            thanks_border,
            text="Thank you for taking the quiz!",
            font=("Arial", 14, "bold"),
            fg=FG_TEXT,
            bg=GREY,
            padx=18,
            pady=10
        )
        thanks.pack(padx=2, pady=2)

        # RESTART BUTTON
        btn_border = tk.Frame(self.frame, bg=BLUE)
        btn_border.pack(pady=(0, 20))

        restart_btn = tk.Button(
            btn_border,
            text="Restart Quiz",
            font=("Arial", 14, "bold"),
            fg=FG_TEXT,
            bg=GREY,
            activebackground=GREY,
            activeforeground=FG_TEXT,
            relief="flat",
            bd=0,
            padx=24,
            pady=12,
            command=self._handle_restart
        )
        restart_btn.pack(padx=2, pady=2)

        # Optional hover effect (lighten grey slightly)
        def _hover_in(e):  restart_btn.configure(bg="#6A6464")
        def _hover_out(e): restart_btn.configure(bg=GREY)
        restart_btn.bind("<Enter>", _hover_in)
        restart_btn.bind("<Leave>", _hover_out)

    # Action
    def _handle_restart(self):
        self.hide()
        if callable(self.on_restart):
            self.on_restart()

    # Control
    def show(self):
        self.frame.pack(fill="both", expand=True)

    def hide(self):
```

```python
        self.frame.pack_forget()

from tkinter import messagebox

class MessageService:
    def warn_no_selection(self):
        messagebox.showwarning("Warning", "Pick an option.")

    def show_correct_with_explanation(self, explanation: str):
        messagebox.showinfo("Correct ✅", f"That’s right!\n\nWhy:\n{explanation}")

    def show_incorrect_with_explanation(self, explanation: str, correct_text: str):
        messagebox.showinfo(
            "Incorrect ❌",
            f"The correct answer is: {correct_text}\n\nWhy:\n{explanation}"
        )

    def show_final_score(self, score, total):
        messagebox.showinfo("Done", f"You scored {score} out of {total}.")


#15 databricks/ data modelling quesions that include:
# - The quesion,
# - Multiple choice options,
# - Which is the correct option
# - Added an extra section that explains why that answer is correct/incorrect, gives explaination.

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."
},
{
"question": "Q2: What is the main purpose of the Gold layer in the Medallion Architecture?",
"options": ["Store raw data as-is", "Store cleaned, enriched, analytics-ready data", "Store machine learning features only", "Archive historical data"],
        "answer_index": 1,
        "explanation": "Gold tables are finalized, conformed datasets aligned to business logic—ready for dashboards and semantic models."
},
{
"question": "Q3: For Data Analysts, why is Delta Lake important?",
"options": ["It encrypts dashboards", "It is needed for machine learning only", "It enables ACID transactions, versioning, and reliable queries", "It automatically builds Power BI models"],
        "answer_index": 2,
        "explanation": "Delta Lake brings ACID reliability, schema enforcement, and time travel, which keeps analytics consistent and auditable."
```

```json
},
{
"question": "Q4: What advantage does Databricks SQL offer to Data Analysts?",
"options": ["A SQL workspace with dashboards and a BI-friendly query editor", "Ability to train deep learning models", "Management of Kubernetes clusters", "Running RPA automations"],
        "answer_index": 0,
        "explanation": "Databricks SQL provides a familiar SQL editor, dashboards, and endpoints tailored to BI-style querying."
},
{
"question": "Q5: Which Medallion layer should contain conformed dimensions and fact tables?",
"options": ["JSON Archive", "Silver", "Bronze", "Gold"],
        "answer_index": 3,
        "explanation": "Conformed, business-aligned dimensions/facts belong in Gold for stable consumption by BI tools like Power BI."
},
{
"question": "Q6: In analytics modelling, what is a 'Fact Table'?",
"options": ["A table of unstructured files", "A table containing business events or measurements", "A table containing lookup values", "A table containing only binary data"],
        "answer_index": 1,
        "explanation": "Fact tables store measurable events (e.g. Provider_sk or Leaner_sk) that aggregate along related dimension attributes."
},
{
"question": "Q7: In a star schema, Dimension Tables usually contain:",
"options": ["System logs", "Descriptive attributes (e.g., product name)", "Only numeric fields", "Duplicate uncleaned fields"],
        "answer_index": 1,
        "explanation": "Dimensions provide descriptive context (attributes) for slicing and filtering fact measures in BI."
},
{
"question": "Q8: Why are surrogate keys important in a star schema?",
"options": ["They avoid dependency on unstable business keys", "They replace all primary keys", "They are faster for machine learning", "They are required for Power BI visuals to work"],
        "answer_index": 0,
        "explanation": "Surrogate keys decouple analytics from changing or non-unique business keys, keeping relationships stable."
},
{
"question": "Q9: Which Databricks feature is most useful for incremental ingestion into Bronze?",
"options": ["SQL Endpoints", "Jobs", "Auto Loader", "Delta Live Tables"],
        "answer_index": 2,
        "explanation": "Auto Loader efficiently discovers and incrementally ingests new files from cloud storage into Bronze."
},
{
"question": "Q10: What is the best layer to join tables for star schema modelling?",
"options": ["Gold", "None — joins shouldn’t be used", "Silver", "Bronze"],
        "answer_index": 2,
        "explanation": "Silver is where data is cleaned and standardized; preparing conformed joins here simplifies building Gold models."
},
{
"question": "Q11: Why shouldn’t Data Analysts query Bronze directly?",
"options": ["It requires Python", "It costs more", "Data is raw, unvalidated, and may contain duplicates", "It is stored in JSON format"],
        "answer_index": 2,
        "explanation": "Bronze is raw landing—may be incomplete, duplicated, or malformed. Analysts should use Silver/Gold instead."
},
{
"question": "Q12: What is a best practice for Power BI developers when using Databricks as a source?",
"options": ["Avoid star schemas", "Import only Gold tables to keep the model clean and stable", "Load data directly from Bronze for speed", "Build Power BI transformations in DAX, not Databricks"],
        "answer_index": 1,
        "explanation": "Gold tables reflect agreed business logic, minimizing Power BI model complexity and refresh issues."
},
{
"question": "Q13: What transformation typically happens in the Silver layer?",
"options": ["Aggregating business metrics", "Geo-spatial mapping", "KPI calculations for dashboards", "Data cleansing, removing duplicates, standardising fields"],
        "answer_index": 3,
        "explanation": "Silver standardizes and validates—deduplication, type casting, and conformance—so Gold can focus on business logic."
},
{
"question": "Q14: If a Power BI model connects to a Databricks Gold table, what is the main benefit?",
"options": ["It skips scheduled refresh", "It auto-generates ETL pipelines", "The data is already cleaned and model-friendly", "It creates measures automatically"],
        "answer_index": 2,
        "explanation": "Gold reduces downstream transformations—models are simpler, more consistent, and easier to maintain."
},
{
"question": "Q15: Which tool in Databricks can orchestrate ELT pipelines for analysts?",
"options": ["Git Repos", "Delta Live Tables", "Notebook Search", "The File Browser"],
        "answer_index": 1,
        "explanation": "Delta Live Tables (DLT) declaratively manages pipelines, quality, and dependencies for reliable ELT."
}
]

import tkinter as tk # Importing the Tkinter module for the quiz UI
from start_screen import StartScreen # This is importing start_screen for the page that comes before the main quiz
```

```python
from ending_screen import EndScreen # This is importing ending_screen for the page that come after the main quiz
from results import append_result # Function that saves name, score, etc. to a CSV

# The colour palette that matches the figma design.
BG_DARK    = "#1F2630"
WHITE      = "#E2E2E2"
ACCENT     = "#F86153"
GREY       = "#5E5757"
BORDER     = "#000000"
BLUE       = "#4D91EA"

class QuizUI:
    def __init__(self, root, engine, messages):
        self.root = root
        self.engine = engine
        self.messages = messages
        self.selected = tk.IntVar()
        # self.user_name = "Analyst" # if I would like add the feature where if no name is put it just puts "Analyst"

        # Window styling
        self.root.title("Databricks Quiz") # Title
        self.root.configure(bg=BG_DARK) # Backbround colour
        self.root.geometry("900x620") # Where the quiz pops up within your display
        self.root.minsize(800, 560) # Minimum size - users can't minimise its size by a cirtain amount otherwise visuals break

        # Start screen
        self.start_screen = StartScreen(root, on_start=self._on_start_clicked)

        # Quiz frame
        self.quiz_frame = tk.Frame(root, bg=BG_DARK)

        # Question row: "Q1:" in ACCENT + question text in white
        self.question_row = tk.Frame(self.quiz_frame, bg=BG_DARK)
        self.question_prefix = tk.Label(
            self.question_row, text="", font=("Arial", 14, "bold"),
            fg=ACCENT, bg=BG_DARK # This makes sure that the "Q1" part of the text comes up as the accent colour (orange/red)
        )
        self.question_text = tk.Label(
            self.question_row, text="", font=("Arial", 14),
            fg=WHITE, bg=BG_DARK, justify="left", wraplength=760 # Adds the rest of text in the white colour
        )

        # Options buttons that will be places on the left hand side
        self.options_frame = tk.Frame(self.quiz_frame, bg=BG_DARK)
        self.option_buttons = []

        # Submit/Next button, styled like your mock (grey with blue border)
        self.button_border = tk.Frame(self.quiz_frame, bg=BLUE)  # border holder
        self.submit_next_button = tk.Button(
            self.button_border,
            text="Click to submit answer",
            font=("Arial", 13, "bold"),
            fg=WHITE,
            bg=GREY,
            activebackground=GREY,
            activeforeground=WHITE,
            relief="flat",
            bd=0,
            padx=18, pady=10,
            command=self._on_submit_or_next
        )

        # Internal state: whether we’re waiting to submit or to go next
        self.awaiting_submit = True # If set to false then the user is moving to the next question

    # Start the quiz
    def _on_start_clicked(self, name: str):
        self.user_name = name # or "Analyst" # If we would like to add a default name if no name is inputted (however we will add Regex to solve this on start screen)
        self.start_screen.hide() # Hide the start screen now that we are on the main quiz screen

        # Starts displaying the quiz!

        # Pack quiz layout
        self.quiz_frame.pack(fill="both", expand=True)

        # Question row
        self.question_row.pack(padx=16, pady=(20, 14), fill="x")
        self.question_prefix.pack(side="left")
        self.question_text.pack(side="left")

        # Options list
        self.options_frame.pack(padx=16, pady=(0, 16), fill="both", expand=True)

        # Button with blue border
        self.button_border.pack(pady=(0, 10))
        self.submit_next_button.pack(padx=2, pady=2)  # blue frame gives the border

        self.show_question() # Load in the first question!

    # Render a question
    def show_question(self):
        q_index = self.engine.index + 1
        q = self.engine.current_question()

        # Set the colored prefix and question text
        self.question_prefix.config(text=f"Q{q_index}: ")
        self.question_text.config(text=q["question"].split(": ", 1)[-1] if q["question"].startswith("Q") else q["question"]) # Updating the question text

        # Clear old options/ radiobuttons
        for rb in self.option_buttons:
            rb.destroy()
        self.option_buttons.clear()
        self.selected.set(-1)

        # 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)

        # Reset button to "submit" state
        self.awaiting_submit = True
        self.submit_next_button.config(text="Click to submit answer", state="normal")

    # 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 then it just gives this general reason so the code dosen't break
        )
        is_correct = self.engine.check_answer(choice) # This updates the score and results list

        # Explanations
        if hasattr(self.messages, "show_correct_with_explanation") and hasattr(self.messages, "show_incorrect_with_explanation"):
            if is_correct:
                self.messages.show_correct_with_explanation(explanation)
            else:
                self.messages.show_incorrect_with_explanation(explanation, self.engine.correct_answer_text())
        else:
            if is_correct:
                self.messages.show_correct()
            else:
                self.messages.show_incorrect()

       # This sections adds the explainations and changes the wording slightly depanding on if the user got the question correct or incorrect.

        # Switch to "Next question" mode
        self.awaiting_submit = False
        self.submit_next_button.config(text="Next question")

    def _next_step(self):
        # Move to next question or end
        if self.engine.next_question():
            self.show_question()

        else:
        # Save result to CSV (name, score, total, timestamp) right at the end of the quiz loop before we get to the end screen.
            append_result(self.user_name, self.engine.score, len(self.engine.questions))

        # End of quiz: go to EndScreen
            self.quiz_frame.pack_forget()
            self.end_screen = EndScreen(
                self.root,
                self.engine.results,            # list of (question_text, is_correct)
                self.engine.score,
                len(self.engine.questions),
                on_restart=self.restart_to_start
        )


    # Fully return to start screen
    def restart_to_start(self):
        self.engine.restart()
        if hasattr(self, "end_screen") and self.end_screen:
            self.end_screen.hide()
        self.start_screen.show()

# start_screen.py
import tkinter as tk
from tkinter import messagebox
```

```python
import re

#PALETTE KEY
BG_DARK    = "#1F2630"
WHITE      = "#E2E2E2"
ACCENT     = "#F86153"
GREY       = "#5E5757"
BORDER     = "#000000"
BLUE       = "#4D91EA"

# Compile once at module load (fast & clean):
# 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]$")


class StartScreen:

    def __init__(self, root, on_start):
        self.root = root
        self.on_start = on_start  # callback(name: str)

        # MAIN BACKGROUND
        self.frame = tk.Frame(root, bg=BG_DARK)
        self.frame.pack(fill="both", expand=True)

        # Center container
        self.center_frame = tk.Frame(self.frame, bg=BG_DARK)
        self.center_frame.place(relx=0.5, rely=0.5, anchor="center")

        # TITLE
        title_frame = tk.Frame(self.center_frame, bg=BG_DARK)
        title_frame.pack(pady=(0, 20))

        tk.Label(title_frame, text="Welcome to the ",
                 font=("Arial", 20, "bold"),
                 fg=WHITE, bg=BG_DARK).pack(side="left")

        tk.Label(title_frame, text="Databricks",
                 font=("Arial", 20, "bold"),
                 fg=ACCENT, bg=BG_DARK).pack(side="left")

        tk.Label(title_frame, text=" Quiz",
                 font=("Arial", 20, "bold"),
                 fg=WHITE, bg=BG_DARK).pack(side="left")

        # SUBTITLE
        tk.Label(self.center_frame, text="What’s your name?",
                 font=("Arial", 16, "bold"),
                 fg=WHITE, bg=BG_DARK).pack(pady=(10, 8))

        # NAME ENTRY
        self.name_entry = tk.Entry(
            self.center_frame, font=("Arial", 14), width=32,
            fg=WHITE, bg=GREY,
            relief="solid", bd=2,
            highlightbackground=BORDER,
            highlightcolor=BORDER,
            highlightthickness=2,
            insertbackground=WHITE
        )
        self.name_entry.insert(0, "Input name:")
        self.name_entry.pack(ipady=10, pady=(0, 22))

        def _clear_placeholder(e):
            if self.name_entry.get() == "Input name:":
                self.name_entry.delete(0, tk.END)
        self.name_entry.bind("&lt;FocusIn&gt;", _clear_placeholder)

        # DESCRIPTION
        desc_outer = tk.Frame(self.center_frame, bg=BORDER)
        desc_outer.pack(pady=(0, 22))
        desc_inner = tk.Frame(desc_outer, bg=GREY, bd=0)
        desc_inner.pack(padx=2, pady=2)
        self.description_label = tk.Label(
            desc_inner,
            text=(
                "Description: What the quiz is about\n"
                "This quiz covers Databricks concepts and the Medallion Architecture.\n"
                "It is designed for data analysts and Power BI practitioners."
            ),
            font=("Arial", 12),
            fg=WHITE,
            bg=GREY,
            justify="left",
            anchor="nw",
            padx=12, pady=12,
            wraplength=560
        )
        desc_inner.configure(width=584)
        self.description_label.pack(fill="both")

        # START BUTTON
        self.start_btn_border = tk.Frame(self.center_frame, bg=BLUE)
        self.start_btn_border.pack(pady=(0, 6))
        self.start_button = tk.Button(
            self.start_btn_border,
            text="Click to start quiz",
            font=("Arial", 14, "bold"),
            fg=WHITE,
            bg=GREY,
            activebackground=GREY,
            activeforeground=WHITE,
            relief="flat",
            bd=0,
            padx=24, pady=12,
            command=self._handle_start
        )
        self.start_button.pack(padx=2, pady=2)

        def _hover_in(e):  self.start_button.configure(bg="#6A6464")
        def _hover_out(e): self.start_button.configure(bg=GREY)
        self.start_button.bind("&lt;Enter&gt;", _hover_in)
        self.start_button.bind("&lt;Leave&gt;", _hover_out)

        # Pressing Enter starts the quiz
        self.name_entry.bind("&lt;Return&gt;", lambda e: self._handle_start())

    # 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)

    # CONTROL
    def show(self):
        self.frame.pack(fill="both", expand=True)

    def hide(self):
        self.frame.pack_forget()

import csv # Import csv module so it can push the results to a csv
from datetime import datetime # For the date/ time
from pathlib import Path # Where the results will be posted

```

```python
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])

import unittest # Importing the unittest module which I will use for testing
from unittest.mock import patch # For mock testing, used to simulate the pop up windows etc

from quiz_engine import QuizEngine # Importing from quiz_engine for testing
from quiz_questions import QUESTIONS # Importing from quiz_questions testing


class TestSmoke(unittest.TestCase): # Smoke test to make sure that unittest is working (testing the test)

    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 its empty or broken

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, [])

    # Check that when an engine is created, the question index is 0, score is 0, and no results have been stores.
    # verifies that the __init__ function is workign correctly

    def test_correct_answer_increments_score(self): # Test correct answers
        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 that the correct answers work as they should do by confirming:
    # The method returned true
    # The score increased to 1
    # One result was stored

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

    # Gets the answer index and adds one to make sure the answer is incorrect as that is what we are testing for

        result = self.engine.check_answer(wrong)

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

    # Testing incorrect answers work as they should do by confirming:
    # Returned false
    # Score did not go up
    # The results was still stored

    def test_next_question_works(self): # Tests that it continuously call the ext question until it returns false
        moves = 0
        while self.engine.next_question():
            moves += 1

        self.assertEqual(moves, len(self.engine.questions) - 1)
        self.assertFalse(self.engine.next_question()) # Calls next_question() one more time at the end to ensure it returns false

    # Checks that we can move through all questions and stops at the end

    def test_restart_resets(self): # Tests the restart button has reset the quiz
        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, [])

    # Answers a question
    # Moves to the next question
    # Calls restart and verify's everything has been reset properly, Question index, Score and user results.

if __name__ == "__main__":
    unittest.main()

```