Portfolio Evidence: Python Multiple-Choice Quiz Game Lesson Plan Extract

Subject: IGCSE Computer Science
Topic: Python Multiple-Choice Quiz Game
Class: KS3, 14 students
Focus: Adaptive teaching through modelling, scaffolding, pair programming, challenge tasks and differentiated exit-ticket responses.
This lesson shows how I supported students with different starting points in an IGCSE Computer Science programming lesson. Students built a Python multiple-choice quiz game using variables, input, conditional statements, comparison operators and score tracking. The lesson used recall, live modelling, deliberate mistake correction, pair programming, targeted questioning, extension tasks and an adaptive exit ticket to make the learning accessible while maintaining challenge.

Learning Context	Students have already met basic Python concepts including input(), print(), variables, if/else, ==, and updating a variable such as attempts = attempts + 1. This lesson applies those ideas in a small software development task: creating a multiple-choice quiz game. The lesson builds from recall, to guided modelling, to pair programming, and finally to independent explanation.	
Aims for learning	**Learning Objectives:**	**Evidence of Learning:**
	By the end of the lesson, students will be able to: 1. Work as a software development pair to build a quiz game. 2. Use conditions to check user answers. 3. Track scores using variables. 4. Explain their code using correct Computer Science terminology.	By the end of the lesson, most students will have created a quiz game with at least 3 questions, multiple-choice answers, if statements, score tracking, and feedback for correct or incorrect answers. Some students will add challenge features such as accepting uppercase/lowercase answers using .lower(), adding more questions, bonus points, final score messages, functions, loops, or replay options.
Anticipated misconceptions and planned responses	*[Misconception check]* Students may confuse = and ==. I will address this during recall and modelling by asking students to explain the difference between assigning a value and comparing two values. Some students may also forget to initialise score = 0, update the score only when the answer is correct, or use indentation correctly inside an if/else block. During live modelling, I will deliberately include small mistakes and ask students to identify and correct them.	
Adaptive and inclusive teaching	*[Adaptive support]* Support will be provided through recall questions, live modelling, shared correction of mistakes, printed task instructions, and pair programming. Students who need more structure can follow the modelled example and complete the core task. More confident students can attempt challenge tasks such as .lower(), bonus questions, final score messages, functions, or loops. The exit ticket uses adaptive teaching strategies by allowing students to select from tasks with different levels of support and challenge.	

Element	**Time**	**What is the learning focus?** What will students be learning – linked to Learning Objectives.	**What will be happening in the classroom?** Outline specific actions you need to take as a teacher as well as what students will be doing.	**How will I check they are learning (formative assessment)?** What strategies will you use?	**Resources / Environment / Classroom Management**
Start of Lesson Routine (creating a climate for learning)	0–3 min	Start routine and objectives	Teacher welcomes students, settles class, introduces lesson title and learning objectives. Students listen and understand the purpose of the lesson.	Check attention through brief questioning: “What will we build today?”	Slides, projector.
Introduction	3–8 min	Code recall	*[Adaptive support]* Teacher shows the password program and asks: “What does input() do?”, “What does == mean?”, “What does attempts = attempts + 1 do?”, “What will be printed?” Students answer and explain.	Cold calling and mini-whiteboards. Follow-up questions to check understanding of variables, comparison, and branching.	Recall code slide, mini-whiteboards
Main body of lesson	8–11 min 11–20 min 20–22 min 22–34 min	Introduce task Guided modelling Transition to pair work Pair programming task	Teacher shows the quiz game description and requirements. Students identify the key success criteria: at least 3 questions, choices, user input, if statements, score, final result, feedback. *[Modelled example]* Teacher models the first quiz question with the class. Teacher asks students what code should come next, *[Misconception check]* deliberately adds small mistakes, and asks students to correct them. Students help build the model answer. *[Peer support]* Teacher explains pair programming expectations: one computer open, one driver, one navigator, both must discuss. Teacher gives paper instructions and challenge tasks. *[Adaptive support]* Students create the quiz game in pairs. Teacher circulates, checks code, asks probing questions, and	Ask students to restate one requirement in their own words. *[Misconception check]* Live questioning: “Where should the score start?”, “Why do we use if answer == "b"?”, “When should the score increase?” Check instructions: “What should your pair do first?” Teacher observes pairs and asks: “Show me where the score changes”, “Explain this if statement”, “What happens if the answer is wrong?”	Slides and printed task sheet Projected Python editor / slide model Printed requirements One computer per pair, task sheet

			supports debugging. Early finishers attempt challenge tasks.		
Plenary	34–38 min 38–40 min	Exit ticket Plenary	*[Adaptive exit ticket]* Students work independently. Part 1 is self-assessment. Part 2: students choose A, B, or C depending on confidence: fill blanks, explain code, or improve code. Teacher asks 2–3 students to share one thing they learned or one problem they solved. Teacher reinforces key vocabulary: variable, assignment, comparison operator, conditional branching.	Collect exit tickets to check understanding of terminology and code logic. Final oral check: "What is the difference between = and ==?"	Exit ticket paper Board / slide

Lesson evaluation

The lesson went quite well overall and the students were responsive and engaged throughout the activities. Most students were able to successfully work in pairs to create a multiple-choice quiz game using conditions, variables, and score tracking. The guided modelling and pair programming structure helped students remain focused and actively discuss their ideas before coding.

The recall activity at the beginning of the lesson helped students revisit key concepts such as input(), ==, and variable updates. During questioning, several students were able to explain the difference between assigning and comparing values. A student participated actively during this phase and correctly answered questions related to the initialization of variables and the use of variables in the quiz program.

Students also showed progress during the pair programming activity. Many students were able to debug errors independently after teacher questioning and prompting. Another student successfully completed some of the extension challenges, showing confidence in applying more advanced features and demonstrating secure understanding of the core task.

The exit tickets provided strong evidence of progress. Most students were able to explain code logic using correct Computer Science terminology. The adaptive structure of the exit ticket allowed students to select tasks appropriate to their confidence level while still demonstrating understanding.

Overall, the learning objectives were met, as most students successfully created a functioning quiz game and demonstrated improved confidence in explaining their code and reasoning.

What I learned and next steps

This lesson reinforced the importance of combining clear modelling with opportunities for students to think and debug independently. Programming can be challenging because students need to understand the logic of the code, use precise syntax, and explain what each line does. The live modelling helped students see how a quiz program could be built step by step, while deliberate mistakes gave them an opportunity to identify and correct common misconceptions.

Pair programming was effective because it allowed students to discuss their ideas before writing code. The driver and navigator structure helped students support each other, and many students were able to debug errors through questioning rather than simply being given the answer. This showed that adaptive teaching gave students enough structure to access the task and then encouraging them to become more independent.

The extension tasks were useful for maintaining challenge for more confident students. Some students were ready to improve their programs by handling uppercase and lowercase

answers, adding more questions, creating bonus points, or experimenting with functions and loops. This helped ensure that students who completed the core task early continued to think deeply and apply their understanding.

The exit ticket provided useful evidence of learning because students could choose from different response options. Some students used the scaffolded fill-in-the-blanks task to consolidate key terminology, while others explained code logic or suggested improvements. This helped me see which students were secure with variables, comparison operators and conditional branching, and which students needed more support with explanation.

For future teaching, I would continue using modelling, pair programming and tiered exit tickets, but I would strengthen student talk during the coding task. I would encourage students to ask more questions independently and use more English when explaining code to their partner. I would also include a short mid-task checkpoint where each student explains one part of their program, so that individual understanding is visible even during pair work.