HyperionDev

IO Operations

Model Answer Approach

Visit our website

Auto-graded task 1

The solution starts by opening the **DOB.txt** file in read mode with UTF-8 encoding to ensure it can handle various file formats. Using a `for` loop, it reads each line of the file and divides it into segments using Python's `split()` method. By slicing each split line to access specific elements, the solution can extract the names and dates of birth, storing them in separate lists (`names` and `dates_of_births`). It then organises and prints these values under the headings (“Name” and “Birthdate”) to clearly present the information.

This approach is well-suited for the practical task, as it effectively demonstrates reading and processing structured data from a text file using Python's file-handling capabilities. By systematically iterating through each line of DOB.txt, extracting and organising names and dates of birth into separate sections, it showcases efficient file input-output operations and string manipulation techniques. This method meets the task requirements while providing a clear and organised presentation of the file's contents.

However, implementing this solution may encounter challenges, such as handling unexpected file formats or errors due to file access issues. It's crucial to anticipate and handle exceptions like **FileNotFoundError** or encoding errors that may occur during file access or reading. Ensuring proper file format validation and implementing robust error handling mechanisms is essential for creating reliable file processing applications in Python.

Auto-graded task 2

The solution for task 2 begins by prompting the user to input the number of students who are registering, ensuring dynamic interaction based on the user input. This initial step is crucial, as it determines the number of iterations for collecting each student's ID. Utilising a `for` loop based on the number of students provided, the program then continues to prompt the user for each student's ID. This ensures that the program can handle multiple student IDs, accommodating varying numbers of students as specified by the user.

Following the inputs, the program opens the **reg_form.txt** file in `write` mode, using UTF-8 encoding to ensure compatibility across different file encodings. For each student ID entered by the user, the program writes it to the file. String formatting ensures that each student ID is followed by a placeholder sequence of dots (“.....”), serving as a visual marker for where the student should sign. This approach is well-suited for managing user input and performing file-writing tasks efficiently, providing a clear and structured solution to the specified task.

However, potential pitfalls may include handling unexpected input, such as non-numeric entries for student IDs, which would require additional validation

mechanisms. Ensuring robust error handling and validation routines can enhance the module's reliability, improving the user experience and preventing unexpected issues during execution.