Name: Cathy DuPuis
Date: August 27, 2024
Course: IT FDN 110 Su 24 Foundations of Programming Python

Assignment 08 – Creating Applications

Introduction

This document provides an overview of the Python application developed for Module 08, covering the key classes, properties, and validation logic. It also includes a detailed section on error handling, with an emphasis on challenges related to the datetime module. In this assignment, I split out Prof. Root's Python program into separate code modules for the data, processing and presentation classes. I created 3 unit tests also called test harnesses for quality assurance. The program manages employee first and last names, their review date and rating and does some simple error checking to prevent user input errors depending on the data type (numerical or alpha or date). It also allows the user to save the data in a JSON format to a file called EmployeeRatings.

Topic – Key Classes

The assignment required FileProcessor, IO, Person, and Employee classes. The FileProcessor class handles file operations like reading and writing employee data to a file, ensuring data is properly loaded and saved in the correct format. The IO (Input/Output) class manages user interaction, including displaying menus, collecting input, and presenting employee data, and is the interface between the user and the application. Both classes are essential for maintaining data flow and user communication within the application. The 2 Key interaction classes for the user inputs are the Person and Employee classes.

The Person class is the basic personal information, and has the following properties:

- **first_name (str)**: The person's first name.
  - **Validation**: Ensures the name contains only alphabetic characters. If a non-alphabetic character is provided, a ValueError is raised.
- **last_name (str)**: The person's last name.
  - **Validation**: Like the first_name, ensures the name contains only alphabetic characters, raising a ValueError if it doesn't.

The Employee class extends the Person class and adds additional properties for the employee data:

- **review_date (datetime.date)**: The date of the employee's performance review.
  - **Default Value**: Defaults to 1900-01-01 if not there during initialization

    - **Validation**: The review_date property accepts either a datetime.date object or a string in the YYYY-MM-DD format. If a string is provided, it is converted to a datetime.date object.
    - **Error Handling**: Raises a ValueError if the input is not a valid date or a properly formatted string.
- **review_rating (int)**: The rating from the performance review.
    - **Default Value**: Defaults to 3 if not there during initialization
    - **Validation**: Ensures the rating is an integer between 1 and 5. If not, a ValueError is raised.

Topic – Error Handling

General Error Handling

The application includes error handling throughout the code. Specific sections of the code are wrapped in try/except blocks to catch and handle exceptions, and provide detailed error messages to the user. This ensures that the application fails (somewhat) gracefully and provides useful feedback when issues arise.

I also had a lot of practice debugging the test modules as those were prone to error themselves especially the same kind of conversion errors as with the strings to objects handling.

Challenges with datetime.date handling

I had a lot of errors with the handling of datetime.date objects. These errors were really time consuming to figure out and presented significant challenges during the development of this application. The issues when trying to ensure consistency in how dates were managed across different parts of the application. Below are the key difficulties encountered and how they were resolved:

Issue: Incorrect Type Conversion

Problem: The application initially struggled with differentiating between datetime.date objects and strings formatted as dates (e.g., "YYYY-MM-DD"). This led to errors when the datetime.strptime() function was mistakenly applied to objects that were already datetime.date instances. This took a long time to figure out and I needed help with ChatGPT to ask what all the errors meant because my debugging skills are not this advanced.

Here's an example of one of many of the error codes I kept getting:

strptime() argument 1 must be str, not datetime.date

Solution: To address this, the code was changed to include a dedicated method _convert_to_date() within the Employee class. This method checks the type of the review_date value:

If it is already a datetime.date object, it is returned as-is.

If it is a string, it is converted to a datetime.date object using datetime.strptime() (Figure 1 and Figure 2).

```
# Tests that an Employee object is correctly initialized with valid data: first name, last name, review date, and review rating.
# Confirms that review_date is properly converted to a datetime.date object and that all properties are correctly assigned.

class TestEmployee(unittest.TestCase):

    def test_employee_initialization(self):
        employee = Employee( first_name: "Jane", last_name: "Doe", datetime.strptime( __date_string: "2023-08-26", __format: "%Y-%m-%d").date(), review_rating: 4)
        self.assertEqual(employee.first_name, second: "Jane")
        self.assertEqual(employee.last_name, second: "Doe")
        self.assertEqual(employee.review_date, datetime.strptime( __date_string: "2023-08-26", __format: "%Y-%m-%d").date())
        self.assertEqual(employee.review_rating, second: 4)
```

Figure 1. Code showing correct initialization of datetime.strptime in test environment

```
def _convert_to_date(self, value):
    if isinstance(value, date):  # If it's already a date object, return it
        return value
    elif isinstance(value, str):  # If it's a string, convert to date
        return datetime.strptime(value, __format: "%Y-%m-%d").date()
    else:
        raise ValueError("Review date must be a datetime.date object or a string in 'YYYY-MM-DD' format.")
```

Figure 2. Code showing correct conditional for datetime.strptime in main script

Outcome: The solution put the the date conversion logic in one location, ensuring that dates were only converted when necessary and avoiding redundant operations that caused errors between the modules.

Issue: Handling Default Dates

Problem: Setting a default date value (1900-01-01) required a lot of time to figure out all the error codes, too. Eventually the default was consistently applied as a datetime.date object, rather than a string.

Solution: The __init__ method of the Employee class was updated to accept the review_date as a string initially, but immediately convert it to a datetime.date object using the _convert_to_date() method. This ensured that the default date was correctly managed as a datetime.date object throughout the application. (Figure 3)

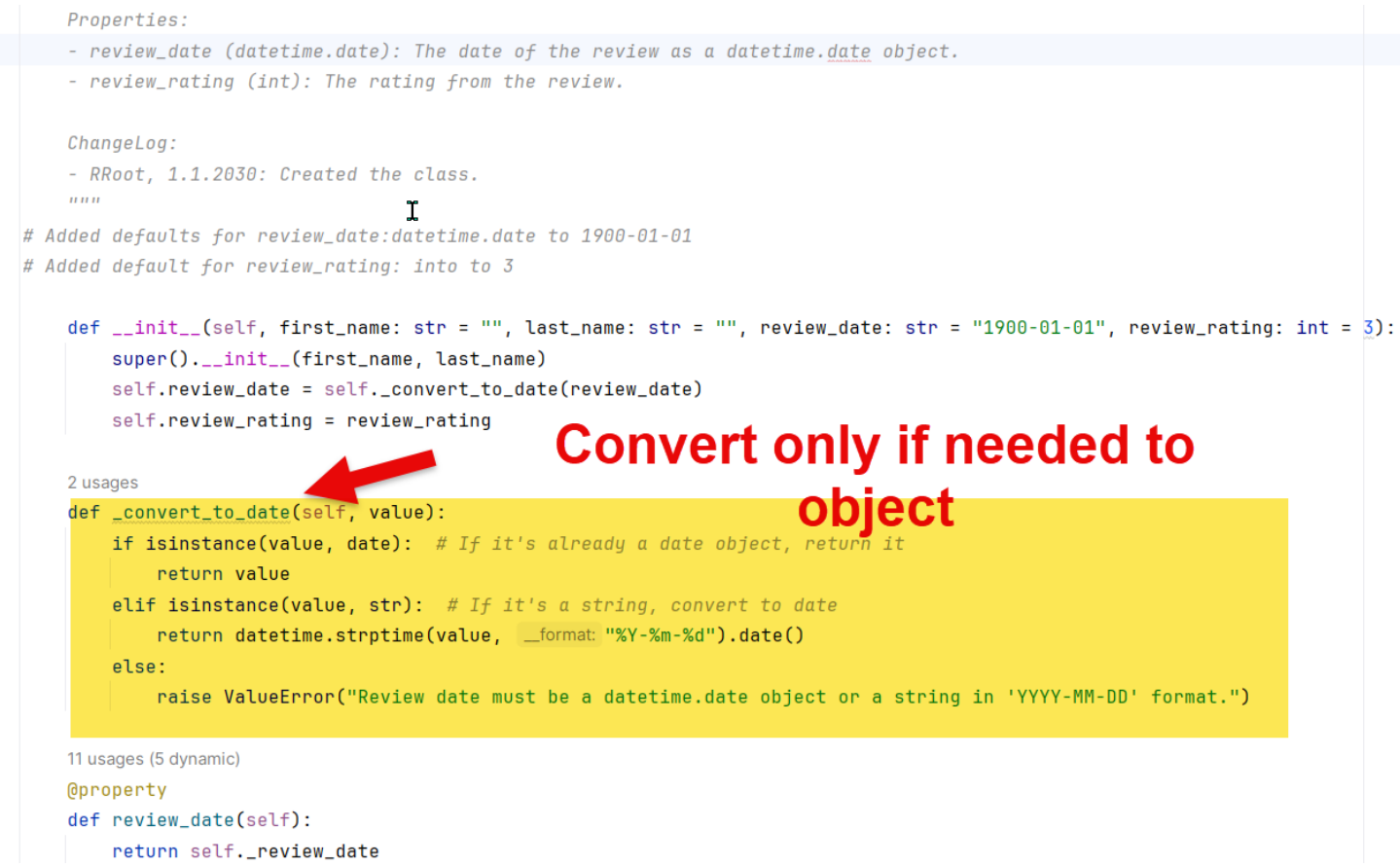

Figure 3. Code showing string defaults for review_date; that get converted to object

I learned that working with datetime objects requires careful attention to data types and conversions -it seems simple, but it is more complicated than it would seem. It is important that date-related operations are only performed on compatible types (e.g., strings should be converted to datetime.date objects before any date-specific operations are applied).

Centralizing conversion logic, as done with the _convert_to_date() method, can greatly reduce the risk of errors and make the code more maintainable and better performing. I also avoids redundancies that throw a lot of errors.

Topic – Testing

There was more testing and debugging to figure out how to make the program work than we have done in the class before. Using menu choice 3, the data gets saved to Enrollements.json and can be opened in a text editor like notepad to show the dictionary list rows (Figure 4).

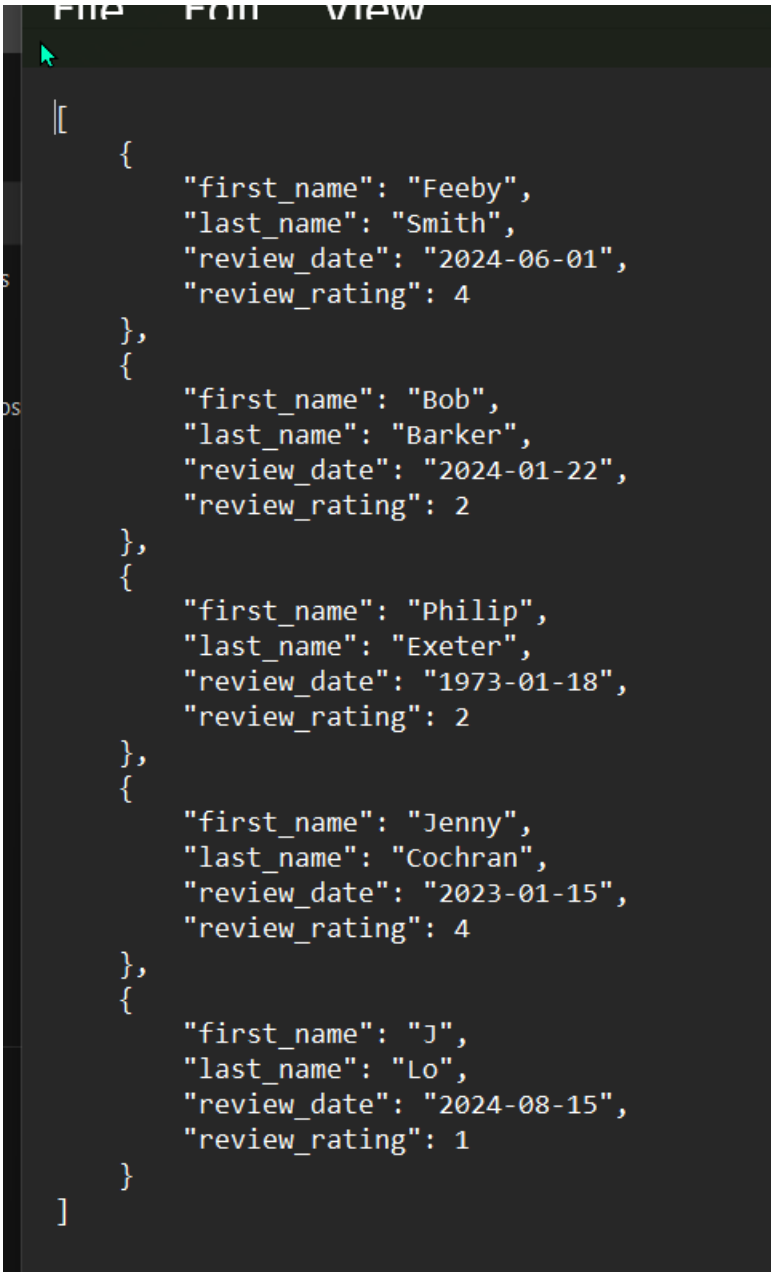

Figure 4. Text file of json employee data (Figure 4).

The program runs correctly in Pycharm and shows all the data of multiple users and exits correctly (Figure 5). The program also successfully prevents a user from entering improper data for the names, date or rating inputs.

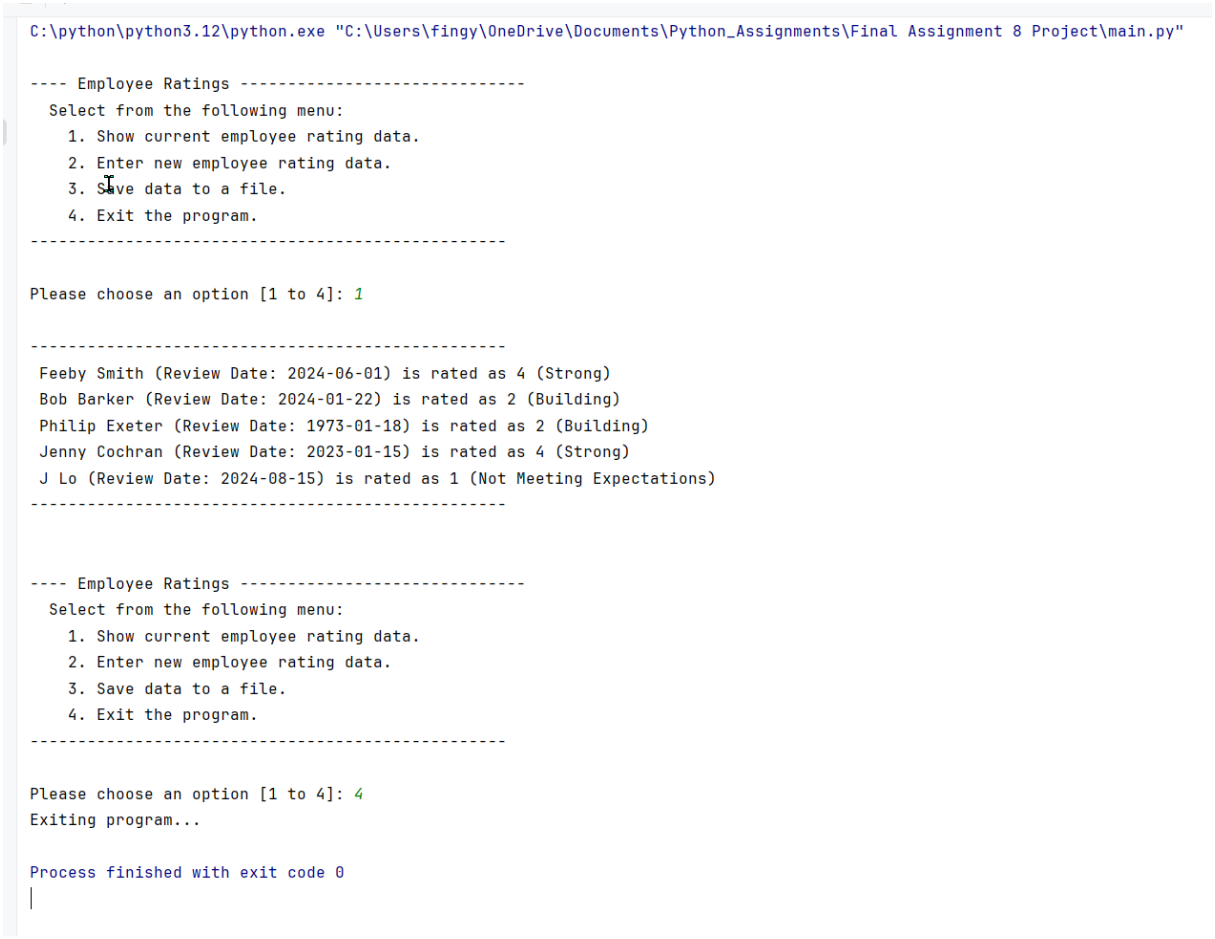

Figure 5. Main program runs smoothly in Pycharm.

Per the assignment, I also tested in the MS Command Prompt and the program ran successfully in that environment (Figure 6). I verified that the program can take new employee first and last name data, their rating date and save data to a file, and handle multiple registrations and that it can display the dictionary data in Notepad with the key and individual string values. I also checked that the simple validation checks were running properly and they were.

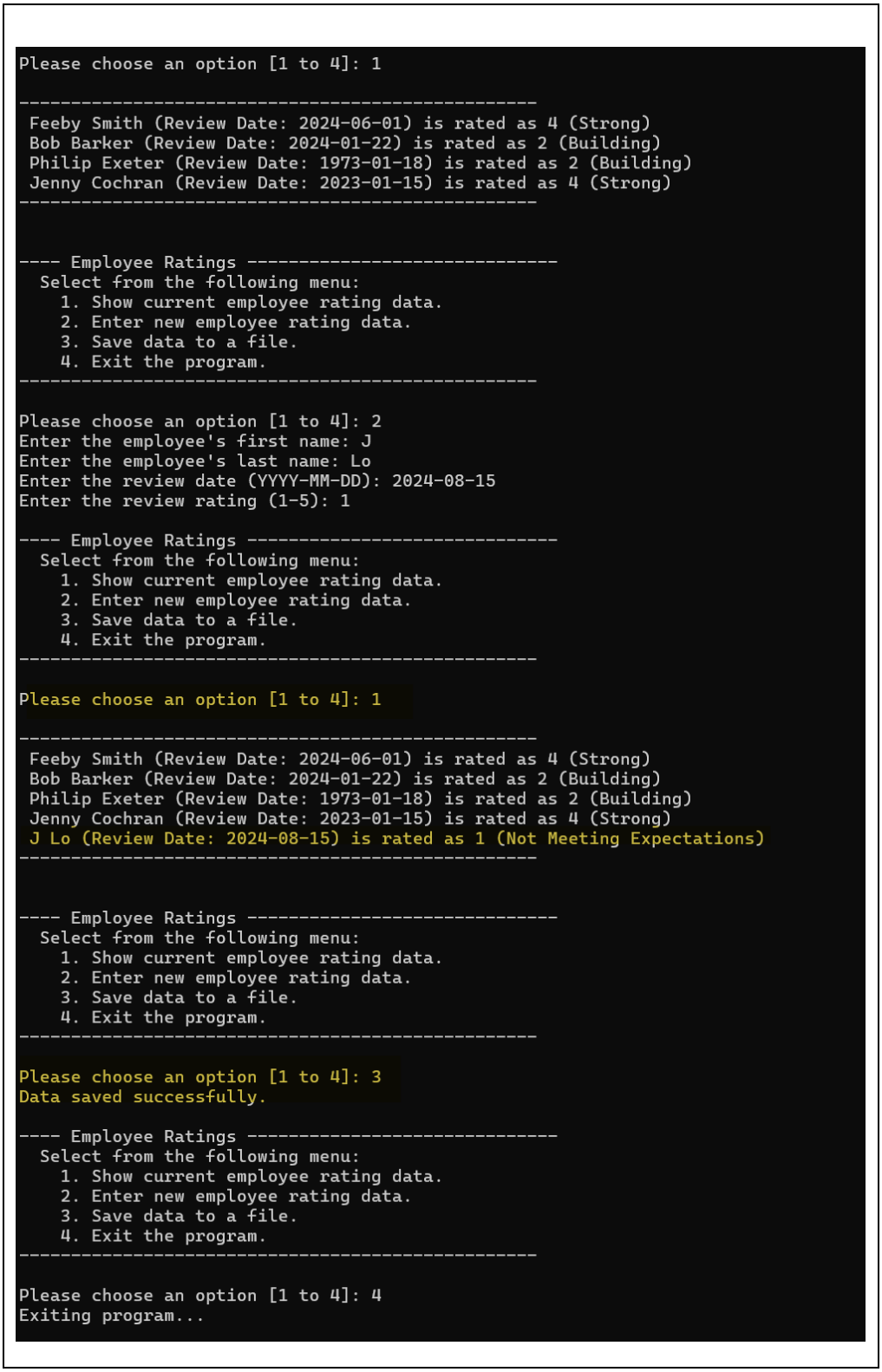

Figure 6. Command prompt working screen shot showing successful program runtime

Also, per the assignment, the 3 modules for testing help to make sure all the different parts of the program run smoothly so that the code behaves as a user would expect. I can import the unittest framework to write and run tests for the presentation, processing and classes. This helps because it is really hard to figure out where the errors are coming from when you have

multiple modules that are being used by the main part of the program. An example of the presentation class test module showing tester data works OK (Figure 7).

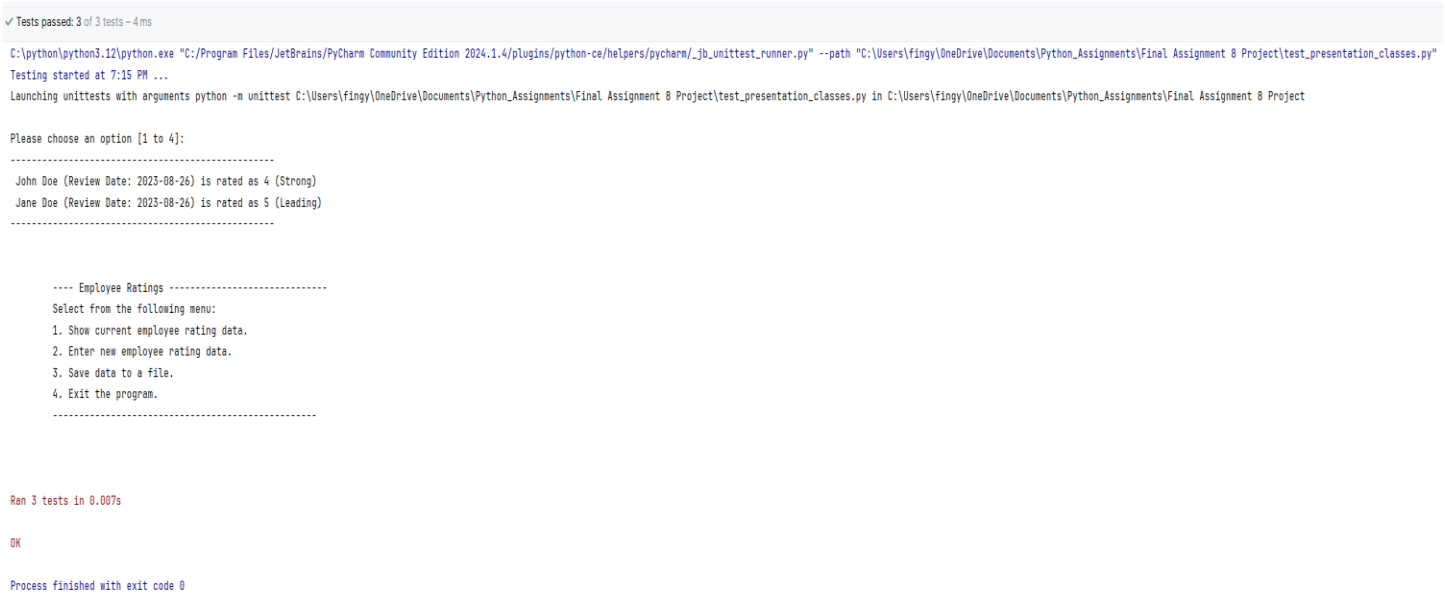

Figure 7. Presentation class test module runs and shows OK.

Summary

This application demonstrates the importance of validation and error handling in Python applications that have multiple modules. The challenges encountered with datetime handling highlighted the need for careful management of data types, especially when dealing with date-related data. The conversion solutions and testing modules that are part of this application ensure that the data remains consistent and that errors are handled so that the user doesn't have a lot of error codes and can have the least amount of surprises.

Github posting: https://github.com/CathyD-UW/IntrotoProg-Python-Mod08.git