Sophia Brooks

06/16/2025

IT FDN 110 A

Assignment #8

https://github.com/SEBrooks25/IntroToProg-Python-Mod08

Creating Applications

Introduction

As programs evolve from small scripts into full applications, organizing code in a clear and maintainable way becomes increasingly important. Module 08 introduces the foundational tools and practices needed to manage this growth, beginning with a clear distinction between programs and applications. While programs are often limited to a single task in a single file, applications require a coordinated structure that supports multiple tasks across several connected files. This module emphasizes the value of code modules, self-contained files that hold related classes and functions, as a way to support modularity and reusability. Using the *import* command, one module can connect to another, allowing functionality to be shared and reused without duplication. Larger projects often involve multiple modules working together, with one designated as the main module that launches the application. To improve readability and avoid naming conflicts, import aliases allow developers to rename modules when importing them. Alongside modular design, this module introduces concepts like architectural planning, UML diagrams, and unit testing to help visualize and verify the application's structure. These tools ensure each component is responsible for a single concern and works predictably with the others. This module lays the groundwork for building robust, scalable applications that are easier to develop, test, and maintain.

Applications

As Python projects grow more complex, understanding the distinction between a program and an application becomes increasingly important even though the terms have been used interchangeably in the past (Figure 1). A *program* typically refers to a smaller, self-contained script designed to perform a limited set of tasks, often through a simple console interface like Python or the PyCharm IDE. These are often written as a single file and focus on automation, data transformation, or specific functionality. An *application* is a larger and more comprehensive software program composed of multiple files, modules, and user-facing features. Applications often include graphical interfaces and require a structured design to remain manageable and reusable over time. As development shifts from isolated scripts to broader solutions, using modules to separate and organize code becomes essential for building flexible, scalable applications.

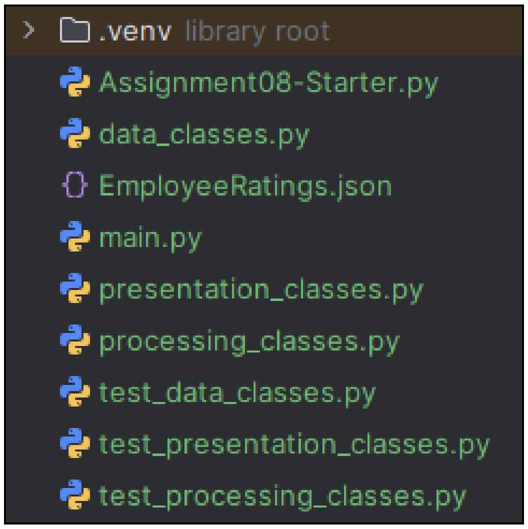
Program	Application
❖ A single file ❖ Minimal Structure ❖ Limited functionality ❖ Console interface ❖ Focused on a single task or goal ❖ Quick to create and run	❖ Multiple Files *and* Modules ❖ Diverse range of features ❖ Requires complex structured design ❖ Utilizes a GUI or Web interface ❖ Supports user interaction and workflow ❖ Organized using design patterns

Figure 1: Table highlighting the differences between a program and an application.

Code Modules

Code modules are individual Python files that contain reusable classes and functions which can be shared across other scripts (Figure 2). Instead of placing all code into a single file, modules allow developers to divide functionality into organized, logical sections. This structure supports the Separation of Concerns pattern by grouping related code into data, processing, or presentation modules. In earlier modules, code was written directly in the main file, but as applications grow more complex, moving those elements into separate module files becomes necessary. For example, placing all data classes into one module and processing functions into another improves clarity and makes the code easier to maintain. This modular design also promotes code reuse, allowing classes and functions to be used in multiple programs without rewriting them. By loading a module into memory, the code inside becomes accessible to any script that imports it.

The *import* command is what connects one module to another, allowing the main script to call classes and functions defined in other modules. Importing a module called *data_classes* gives access to everything inside it, as long as it's referenced using the correct module path. Larger applications often use multiple modules working together, with one script designated as the main module that starts the program. To make long or complex module names easier to work with, developers can assign an alias during import using the "*as*" keyword. This renaming helps avoid typing long module names repeatedly and can prevent naming conflicts. For example, import *presentation_classes* as *pres* shortens all references in the code to simply use *pres*. Using modules and import aliases in this way supports cleaner, more readable code while maintaining the flexibility to scale as the application evolves.



```
> .venv library root
  Assignment08-Starter.py
  data_classes.py
  EmployeeRatings.json
  main.py
  presentation_classes.py
  processing_classes.py
  test_data_classes.py
  test_presentation_classes.py
  test_processing_classes.py
```

Figure 2: Shows list of files indicating different Python files containing reusable classes of code used in Assignment08.

Designing Applications

Designing an application is a structured and ongoing process that begins with understanding the problem and ends with a working solution that meets user needs. It starts with problem analysis, followed by gathering detailed requirements that define what the application should do. Next, a high-level architectural design is created to outline how different components will interact with one another. The development team then selects appropriate tools, such as the programming language, code editor, and version control system, to build the application. After that, the implementation phase begins, where the application is constructed using organized modules that follow design patterns like Separation of Concerns. Once the code is in place, it goes through testing, documentation, and review to ensure it functions correctly and can be maintained over time. This step-by-step approach helps ensure that applications are reliable, easy to understand, and ready to grow with future changes.

Problem analysis and requirements gathering are the first critical steps in designing an application, working together to define what the software needs to accomplish and why. Problem analysis focuses on fully understanding the issue to be solved, including who is affected, what the constraints are, and what the ideal outcome looks like. This step often results in a clear and concise problem statement, which serves as a guide for the rest of the development process by outlining the application's main objective. Once the problem is clearly defined, requirements gathering begins by identifying the specific features, data, and behaviors the application must support. This includes determining what information will be collected, how it will be stored, what tasks the program will perform, and how users will interact with it. From these details, developers define acceptance criteria, which are measurable conditions used to determine whether the application meets its intended goals. Together, problem analysis and requirements gathering provide the foundation for thoughtful application design, ensuring that the software is both useful and aligned with user expectations.

Architectural design provides a high-level blueprint for how the different parts of an application will work together. It defines the main script as the entry point, which coordinates actions between specialized modules. Key components include the Student class, which inherits from a Person class to organize shared and unique attributes, and the FileProcessor class, which manages reading and writing data to a file. The IO class handles user interaction by displaying menus, collecting input, and showing results or error messages. Data is stored in memory as a list of Student objects and validated using property methods to ensure accuracy. The application uses a menu-based structure to guide users through available options and integrates error handling to manage unexpected issues. Altogether, these components form a modular and maintainable structure that aligns with best practices for building scalable applications.

Unified Modeling Language, or UML, is a standardized way of visually representing how different parts of an application interact and relate to one another. It helps developers and stakeholders understand the structure and behavior of software before writing or modifying code. UML is especially useful for identifying design flaws, clarifying requirements, and communicating ideas across a development team. One common type is the *Use Case Diagram* , which shows how users and other systems interact with the application by mapping out typical tasks such as viewing or saving data. These diagrams help ensure that all necessary user actions are accounted for in the design. The *Class Diagram* provides a visual layout of the application's classes, their properties and methods, and the relationships between them, such as inheritance or composition. This is helpful when designing or reviewing class structures and understanding how data and behavior are organized. A *Component Diagram* illustrates how different modules or sections of the code depend on one another, revealing the flow of data and control throughout the system. This type of diagram is especially useful for identifying tight coupling between components that may reduce code flexibility. Lastly, *Composition Diagrams* show how objects created from classes are connected during runtime, helping developers understand object lifecycles and dependencies within the program.

Choosing the right development tools is an important step in turning a design into a working application. This includes selecting a programming language, an integrated development environment, and a version control system that supports the project’s goals and team workflow. In this course, the focus has been on using the Python programming language, the PyCharm IDE for writing and testing code, and GitHub for version control and collaboration. After the tools are selected, the software development step begins, where the application is constructed based on the architectural design and gathered requirements. This phase involves writing the code, structuring it into modules, and building out the planned features one piece at a time. Once the tools are in place, the implementation phase begins by translating the design into actual code, often organized using the Separation of Concerns pattern. Classes and functions are grouped into modules based on their roles, making the code easier to manage and reuse in future projects. This structured approach sets the foundation for building scalable and maintainable applications.

Testing and quality assurance are essential parts of application development that help ensure the program functions as expected and handles errors gracefully. One of the most effective ways to catch issues early is through unit testing, which involves checking individual parts of the code such as functions, methods, or classes in isolation. Test cases are written to simulate different inputs and behaviors, allowing developers to verify that each component responds correctly under various conditions. Python includes a built-in assert statement that can be used to perform simple checks and raise an error if a condition is not met. For more structured testing, Python’s unittest framework provides tools for organizing test cases, comparing expected and actual results, and reporting successes or failures. Running these tests frequently helps catch bugs early, making the codebase more stable and easier to maintain. Another important concept is the __*name*__ system variable, which helps control how and when modules are executed. By checking whether __*name*__ is equal to *"__main__"*, developers can ensure that certain code only runs when a file is executed directly, not when it is imported as a module. This protects the structure of the application and prevents confusion when different parts of the code are used across multiple files. Altogether, these testing techniques support reliable, predictable software and help developers build confidence in their applications as they grow.

Documentation plays a vital role in the application development process by ensuring that software is understandable, maintainable, and usable well beyond its initial release. It supports understanding and learning by providing clear explanations of how the code works, making it easier for new developers or users to get up to speed. In collaborative environments, it helps teams stay aligned and maintain consistent coding practices. Well-written documentation also supports long-term maintenance, allowing developers to troubleshoot and debug with a clear reference to the original design and behavior. It assists in testing by outlining expected results, provides guidance for user onboarding, and may even be required for compliance and auditing purposes in certain industries. As applications evolve, documentation helps with knowledge transfer between developers, supports scaling and extension, and improves communication among all stakeholders. In open-source or collaborative projects, it ensures that external contributors can use and build upon the work effectively. Even legacy systems benefit from strong documentation, making it possible to update or repurpose old code with minimal confusion. Legal protection is another often overlooked benefit, as detailed records of development can help protect intellectual property or resolve disputes. Alongside documentation, the principle of continuous improvement encourages developers to regularly review and refine their software, addressing user feedback and adapting to new requirements over time. Together, documentation and improvement practices ensure that applications remain functional, relevant, and valuable across their entire lifecycle, completing the broader process of thoughtful and sustainable application design.

The Assignment: Creating an Application

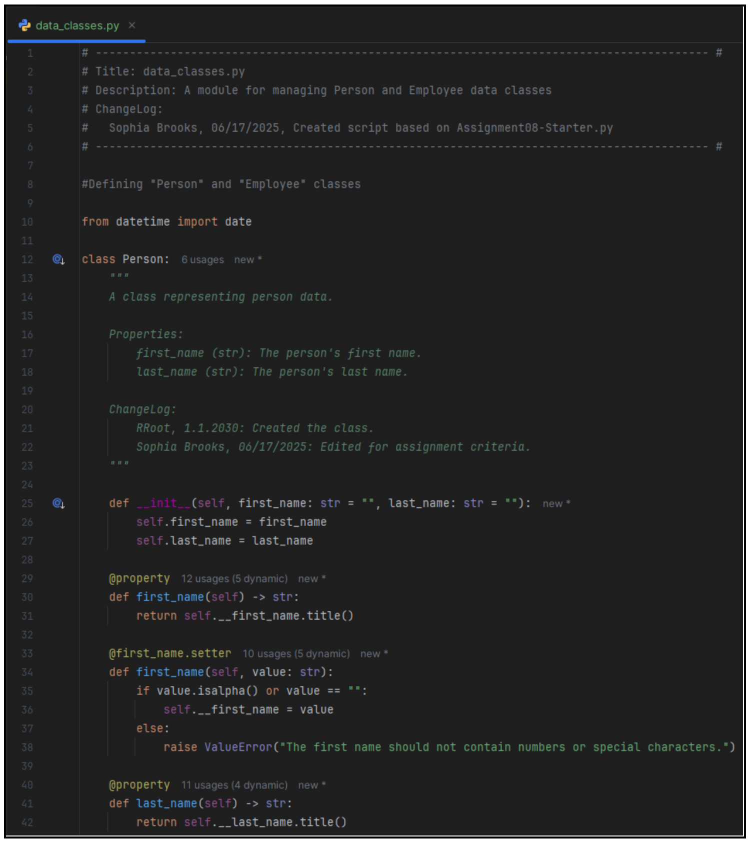
The process of completing the application began by reviewing the provided starter file and carefully comparing it against the acceptance criteria outlined in the assignment instructions. The starter code laid out the functional baseline, but was presented as a single script with all classes, functions, constants, and logic contained in one file. The first step to developing the application focused on planning how to reorganize the code into clearly defined modules. This included separating the data, processing, and presentation logic into distinct files. Before making any structural changes, special attention was given to understanding the role and behavior of each class and function in the original script to ensure nothing was lost during the refactor.

Once the application design was clear, the next step was to extract each major component into its corresponding module. The *Person* and *Employee* classes were placed into a new file called *data_classes.py* (Figures 3.1, 3.2, & 3.3 consecutively), which focused exclusively on defining data structures. Next, the file input and output methods were moved into *processing_classes.py* (Figures 4.1, 4.2, & 4.3), which handled reading and writing employee review data in JSON format. Finally, the user interface logic, displaying menus, receiving input, and printing output was organized into *presentation_classes.py* (Figures 5.1, 5.2, & 5.3). This division followed the Separation of Concerns pattern, allowing each module to handle a specific responsibility. With the modules created, the main script (*main.py* (Figures 6.1 & 6.2)) was rewritten to coordinate the flow between them, importing and calling functions from each module as needed.

During this restructuring, special attention was given to how data flowed between functions and how validation was performed. The class properties in *Employee* were updated to enforce correct formats and value ranges using property decorators and exception handling. One example, the *review_date* property verifies that input is in the correct ISO date format, while the *review_rating* property ensures the value falls between 1 and 5. These validation rules were critical for maintaining data integrity and helped reinforce best practices for designing secure, reliable applications. The *__str__()* methods were refined to return consistently formatted output, which was essential for displaying employee data and saving it to the JSON file.

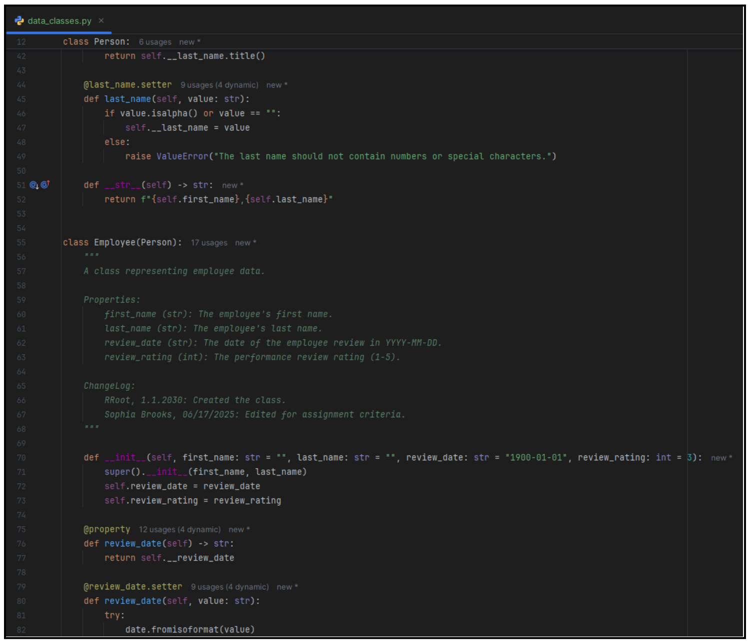
Once the application structure and core logic were finalized, the focus shifted to testing. Three test files were created, each aligning with a corresponding code module; *test_data_classes.py* (Figures 7.1 & 7.2), *test_processing_classes.py* (Figure 8.` & 8.2), and *test_presentation_classes.py* (Figure 9.1 & 9.2). These tests were written using Python's built-in *unittest* framework and were designed to simulate realistic inputs, validate expected outputs, and trigger known errors to verify proper handling. Mocking techniques were used where needed to simulate user input and suppress console output for verification. Writing tests for each major function not only helped confirm that the program behaved as expected, but also provided documentation of how the code was intended to be used.

Completing this application required synthesizing all of the knowledge gained throughout the course into a final, fully functional product (Figures 10.1, 10.2, 10.3, & 10.4). Each decision, from how the modules were named and organized, to how the test cases were written, reflected the principles of modularity, reusability, and maintainability emphasized in earlier lessons. Beyond simply getting the code to run, the focus was on designing an application that could scale, be tested easily, and remain understandable over time. In the end, this structured approach allowed the application to meet both technical and design goals, providing a solid foundation for future development in more complex or collaborative environments.



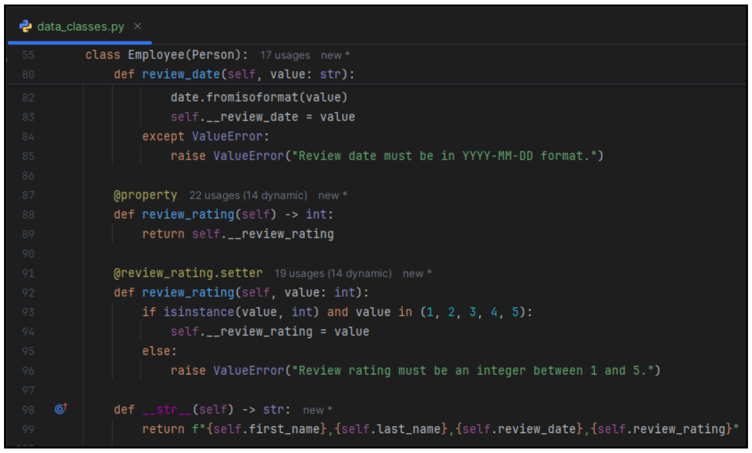
```python
# ------------------------------------------------------------------------------------------ #
# Title: data_classes.py
# Description: A module for managing Person and Employee data classes
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created script based on Assignment08-Starter.py
# ------------------------------------------------------------------------------------------ #

#Defining "Person" and "Employee" classes

from datetime import date

class Person:
    """
    A class representing person data.

    Properties:
        first_name (str): The person's first name.
        last_name (str): The person's last name.

    ChangeLog:
        RRoot, 1.1.2030: Created the class.
        Sophia Brooks, 06/17/2025: Edited for assignment criteria.
    """

    def __init__(self, first_name: str = "", last_name: str = ""):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self) -> str:
        return self.__first_name.title()

    @first_name.setter
    def first_name(self, value: str):
        if value.isalpha() or value == "":
            self.__first_name = value
        else:
            raise ValueError("The first name should not contain numbers or special characters.")

    @property
    def last_name(self) -> str:
        return self.__last_name.title()
```

Figure 3.1



```python
class Person:
        return self.__last_name.title()

    @last_name.setter
    def last_name(self, value: str):
        if value.isalpha() or value == "":
            self.__last_name = value
        else:
            raise ValueError("The last name should not contain numbers or special characters.")

    def __str__(self) -> str:
        return f"{self.first_name},{self.last_name}"


class Employee(Person):
    """
    A class representing employee data.

    Properties:
        first_name (str): The employee's first name.
        last_name (str): The employee's last name.
        review_date (str): The date of the employee review in YYYY-MM-DD.
        review_rating (int): The performance review rating (1-5).

    ChangeLog:
        RRoot, 1.1.2030: Created the class.
        Sophia Brooks, 06/17/2025: Edited for assignment criteria.
    """

    def __init__(self, first_name: str = "", last_name: str = "", review_date: str = "1900-01-01", review_rating: int = 3):
        super().__init__(first_name, last_name)
        self.review_date = review_date
        self.review_rating = review_rating

    @property
    def review_date(self) -> str:
        return self.__review_date

    @review_date.setter
    def review_date(self, value: str):
        try:
            date.fromisoformat(value)
```

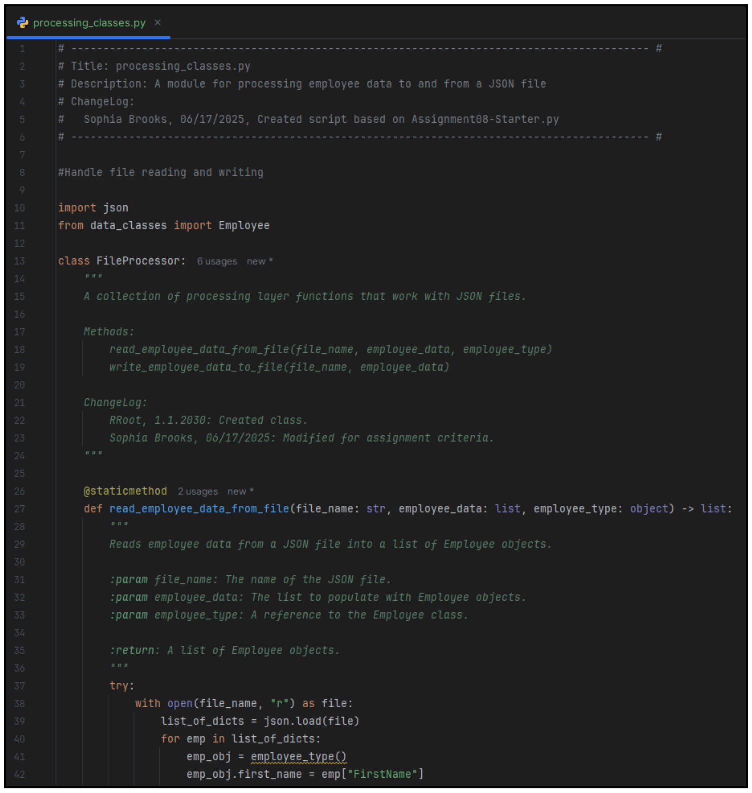
Figure 3.2



```python
    class Employee(Person):
        def review_date(self, value: str):
                date.fromisoformat(value)
                self.__review_date = value
            except ValueError:
                raise ValueError("Review date must be in YYYY-MM-DD format.")

        @property
        def review_rating(self) -> int:
            return self.__review_rating

        @review_rating.setter
        def review_rating(self, value: int):
            if isinstance(value, int) and value in (1, 2, 3, 4, 5):
                self.__review_rating = value
            else:
                raise ValueError("Review rating must be an integer between 1 and 5.")

        def __str__(self) -> str:
            return f"{self.first_name},{self.last_name},{self.review_date},{self.review_rating}"
```

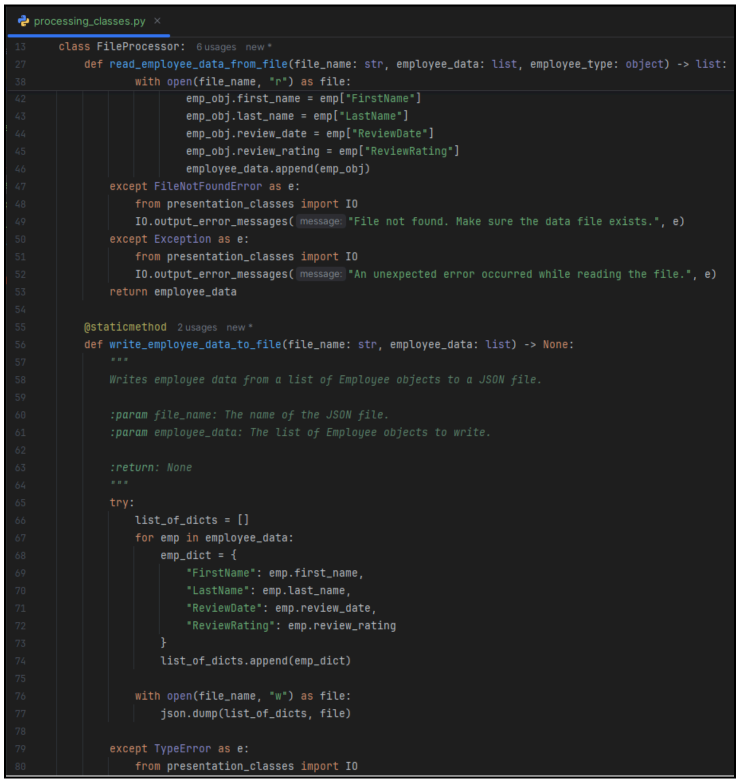
Figure 3.3

Figures 3.1, 3.2, & 3.3: Showing the module script for *data_classes.py* containing the class data for *Person* and *Employee(Person).*



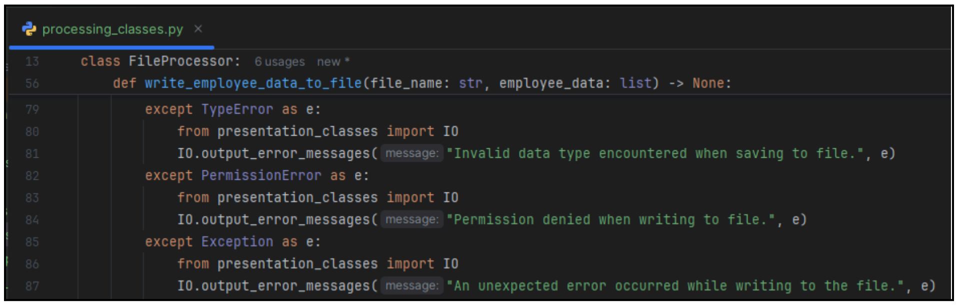
```python
# ----------------------------------------------------------------------------------------- #
# Title: processing_classes.py
# Description: A module for processing employee data to and from a JSON file
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created script based on Assignment08-Starter.py
# ----------------------------------------------------------------------------------------- #

#Handle file reading and writing

import json
from data_classes import Employee

class FileProcessor:
    """
    A collection of processing layer functions that work with JSON files.

    Methods:
        read_employee_data_from_file(file_name, employee_data, employee_type)
        write_employee_data_to_file(file_name, employee_data)

    ChangeLog:
        RRoot, 1.1.2030: Created class.
        Sophia Brooks, 06/17/2025: Modified for assignment criteria.
    """

    @staticmethod
    def read_employee_data_from_file(file_name: str, employee_data: list, employee_type: object) -> list:
        """
        Reads employee data from a JSON file into a list of Employee objects.

        :param file_name: The name of the JSON file.
        :param employee_data: The list to populate with Employee objects.
        :param employee_type: A reference to the Employee class.

        :return: A list of Employee objects.
        """
        try:
            with open(file_name, "r") as file:
                list_of_dicts = json.load(file)
                for emp in list_of_dicts:
                    emp_obj = employee_type()
                    emp_obj.first_name = emp["FirstName"]
```

Figure 4.1



```
processing_classes.py ×
    class FileProcessor:  6 usages  new *
        def read_employee_data_from_file(file_name: str, employee_data: list, employee_type: object) -> list:
                with open(file_name, "r") as file:
                        emp_obj.first_name = emp["FirstName"]
                        emp_obj.last_name = emp["LastName"]
                        emp_obj.review_date = emp["ReviewDate"]
                        emp_obj.review_rating = emp["ReviewRating"]
                        employee_data.append(emp_obj)
            except FileNotFoundError as e:
                from presentation_classes import IO
                IO.output_error_messages( message: "File not found. Make sure the data file exists.", e)
            except Exception as e:
                from presentation_classes import IO
                IO.output_error_messages( message: "An unexpected error occurred while reading the file.", e)
            return employee_data

        @staticmethod  2 usages  new *
        def write_employee_data_to_file(file_name: str, employee_data: list) -> None:
            """
            Writes employee data from a list of Employee objects to a JSON file.

            :param file_name: The name of the JSON file.
            :param employee_data: The list of Employee objects to write.

            :return: None
            """
            try:
                list_of_dicts = []
                for emp in employee_data:
                    emp_dict = {
                        "FirstName": emp.first_name,
                        "LastName": emp.last_name,
                        "ReviewDate": emp.review_date,
                        "ReviewRating": emp.review_rating
                    }
                    list_of_dicts.append(emp_dict)

                with open(file_name, "w") as file:
                    json.dump(list_of_dicts, file)

            except TypeError as e:
                from presentation_classes import IO
```

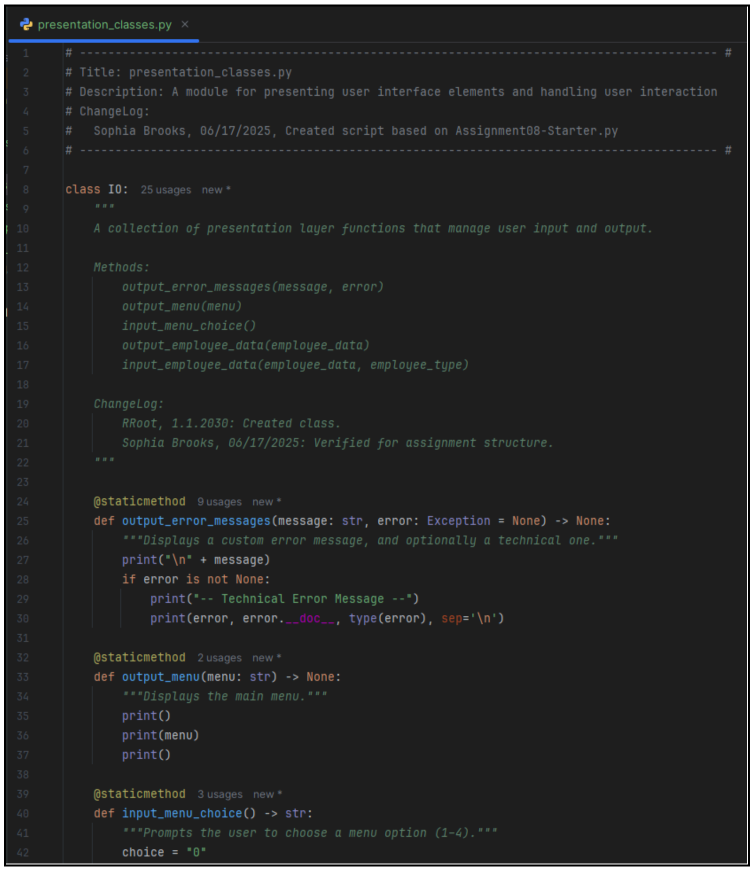
Figure 4.2



```
processing_classes.py ×
    class FileProcessor:  6 usages  new *
        def write_employee_data_to_file(file_name: str, employee_data: list) -> None:
            except TypeError as e:
                from presentation_classes import IO
                IO.output_error_messages( message: "Invalid data type encountered when saving to file.", e)
            except PermissionError as e:
                from presentation_classes import IO
                IO.output_error_messages( message: "Permission denied when writing to file.", e)
            except Exception as e:
                from presentation_classes import IO
                IO.output_error_messages( message: "An unexpected error occurred while writing to the file.", e)
```

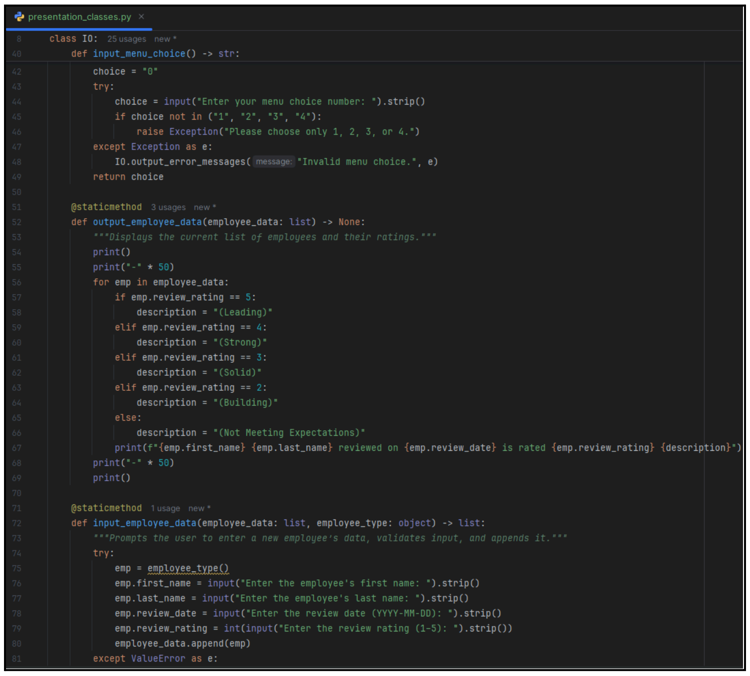
Figure 8.3

Figures 4.1, 4.2, & 4.3: Showing the module script for *processing_classes.py* containing the class methods for *FileProcessor.*



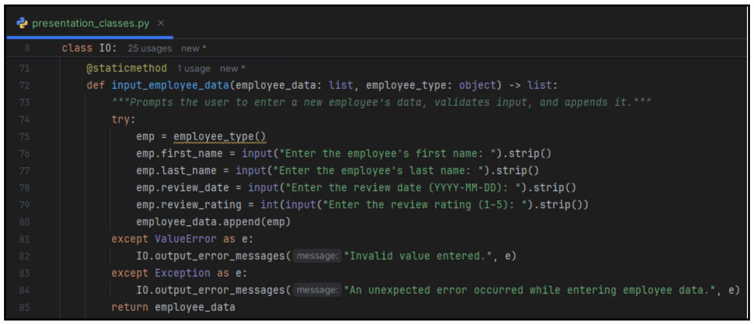
```python
# ---------------------------------------------------------------------------------------- #
# Title: presentation_classes.py
# Description: A module for presenting user interface elements and handling user interaction
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created script based on Assignment08-Starter.py
# ---------------------------------------------------------------------------------------- #

class IO:
    """
    A collection of presentation layer functions that manage user input and output.

    Methods:
        output_error_messages(message, error)
        output_menu(menu)
        input_menu_choice()
        output_employee_data(employee_data)
        input_employee_data(employee_data, employee_type)

    ChangeLog:
        RRoot, 1.1.2030: Created class.
        Sophia Brooks, 06/17/2025: Verified for assignment structure.
    """

    @staticmethod
    def output_error_messages(message: str, error: Exception = None) -> None:
        """Displays a custom error message, and optionally a technical one."""
        print("\n" + message)
        if error is not None:
            print("-- Technical Error Message --")
            print(error, error.__doc__, type(error), sep='\n')

    @staticmethod
    def output_menu(menu: str) -> None:
        """Displays the main menu."""
        print()
        print(menu)
        print()

    @staticmethod
    def input_menu_choice() -> str:
        """Prompts the user to choose a menu option (1-4)."""
        choice = "0"
```

Figure 5.1



```python
class IO:
    def input_menu_choice() -> str:
        choice = "0"
        try:
            choice = input("Enter your menu choice number: ").strip()
            if choice not in ("1", "2", "3", "4"):
                raise Exception("Please choose only 1, 2, 3, or 4.")
        except Exception as e:
            IO.output_error_messages("Invalid menu choice.", e)
        return choice

    @staticmethod
    def output_employee_data(employee_data: list) -> None:
        """Displays the current list of employees and their ratings."""
        print()
        print("-" * 50)
        for emp in employee_data:
            if emp.review_rating == 5:
                description = "(Leading)"
            elif emp.review_rating == 4:
                description = "(Strong)"
            elif emp.review_rating == 3:
                description = "(Solid)"
            elif emp.review_rating == 2:
                description = "(Building)"
            else:
                description = "(Not Meeting Expectations)"
            print(f"{emp.first_name} {emp.last_name} reviewed on {emp.review_date} is rated {emp.review_rating} {description}")
        print("-" * 50)
        print()

    @staticmethod
    def input_employee_data(employee_data: list, employee_type: object) -> list:
        """Prompts the user to enter a new employee's data, validates input, and appends it."""
        try:
            emp = employee_type()
            emp.first_name = input("Enter the employee's first name: ").strip()
            emp.last_name = input("Enter the employee's last name: ").strip()
            emp.review_date = input("Enter the review date (YYYY-MM-DD): ").strip()
            emp.review_rating = int(input("Enter the review rating (1-5): ").strip())
            employee_data.append(emp)
        except ValueError as e:
```

Figure 5.2

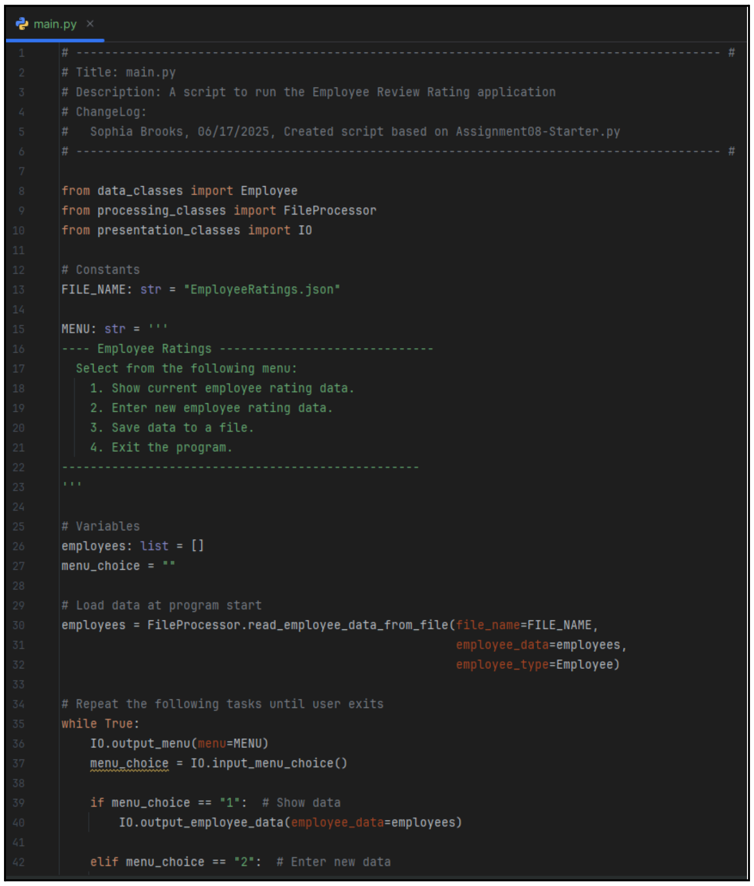


presentation_classes.py

```python
class IO:
...
    @staticmethod
    def input_employee_data(employee_data: list, employee_type: object) -> list:
        """Prompts the user to enter a new employee's data, validates input, and appends it."""
        try:
            emp = employee_type()
            emp.first_name = input("Enter the employee's first name: ").strip()
            emp.last_name = input("Enter the employee's last name: ").strip()
            emp.review_date = input("Enter the review date (YYYY-MM-DD): ").strip()
            emp.review_rating = int(input("Enter the review rating (1-5): ").strip())
            employee_data.append(emp)
        except ValueError as e:
            IO.output_error_messages("Invalid value entered.", e)
        except Exception as e:
            IO.output_error_messages("An unexpected error occurred while entering employee data.", e)
        return employee_data
```

Figure 5.3

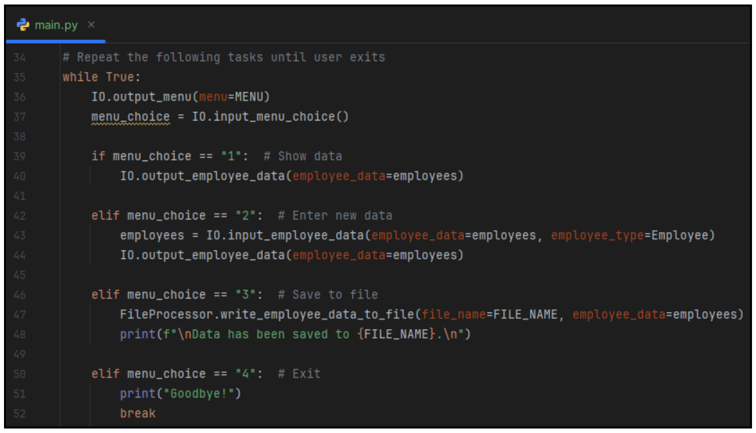
Figures 5.1, 5.2, & 5.3: Showing the module script for *presentation_classes.py* containing the class methods for *IO.*



main.py

```python
# ------------------------------------------------------------------------------------------ #
# Title: main.py
# Description: A script to run the Employee Review Rating application
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created script based on Assignment08-Starter.py
# ------------------------------------------------------------------------------------------ #

from data_classes import Employee
from processing_classes import FileProcessor
from presentation_classes import IO

# Constants
FILE_NAME: str = "EmployeeRatings.json"

MENU: str = '''
---- Employee Ratings ------------------------------
  Select from the following menu:
    1. Show current employee rating data.
    2. Enter new employee rating data.
    3. Save data to a file.
    4. Exit the program.
--------------------------------------------------
'''

# Variables
employees: list = []
menu_choice = ""

# Load data at program start
employees = FileProcessor.read_employee_data_from_file(file_name=FILE_NAME,
                                                       employee_data=employees,
                                                       employee_type=Employee)

# Repeat the following tasks until user exits
while True:
    IO.output_menu(menu=MENU)
    menu_choice = IO.input_menu_choice()

    if menu_choice == "1":  # Show data
        IO.output_employee_data(employee_data=employees)

    elif menu_choice == "2":  # Enter new data
```

Figure 6.1

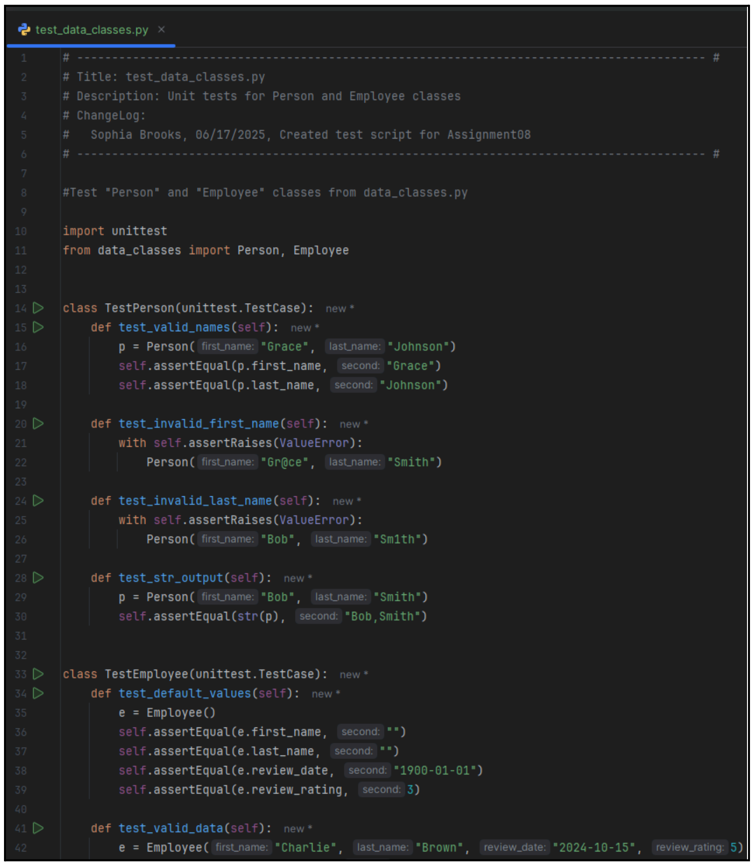


main.py

```python
# Repeat the following tasks until user exits
while True:
    IO.output_menu(menu=MENU)
    menu_choice = IO.input_menu_choice()

    if menu_choice == "1":  # Show data
        IO.output_employee_data(employee_data=employees)

    elif menu_choice == "2":  # Enter new data
        employees = IO.input_employee_data(employee_data=employees, employee_type=Employee)
        IO.output_employee_data(employee_data=employees)

    elif menu_choice == "3":  # Save to file
        FileProcessor.write_employee_data_to_file(file_name=FILE_NAME, employee_data=employees)
        print(f"\nData has been saved to {FILE_NAME}.\n")

    elif menu_choice == "4":  # Exit
        print("Goodbye!")
        break
```

Figure 6.2

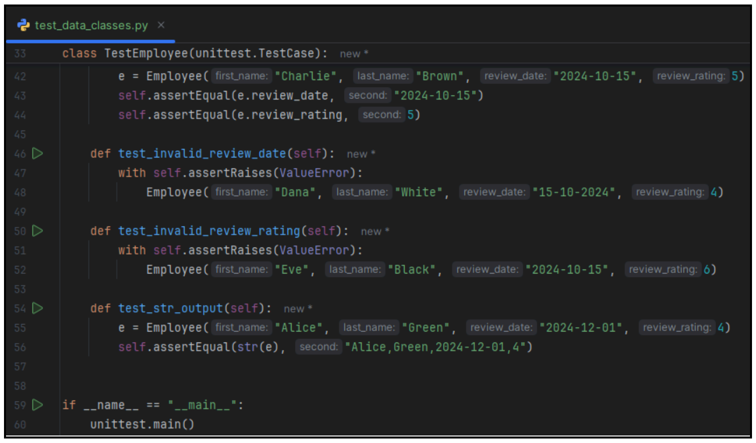
Figures 6.1 & 6.2: Showing the module script for *main.py* designed to run the *Employee Review Rating* application.



test_data_classes.py

```python
# ------------------------------------------------------------------------------------------ #
# Title: test_data_classes.py
# Description: Unit tests for Person and Employee classes
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created test script for Assignment08
# ------------------------------------------------------------------------------------------ #

#Test "Person" and "Employee" classes from data_classes.py

import unittest
from data_classes import Person, Employee


class TestPerson(unittest.TestCase):
    def test_valid_names(self):
        p = Person("Grace", "Johnson")
        self.assertEqual(p.first_name, "Grace")
        self.assertEqual(p.last_name, "Johnson")

    def test_invalid_first_name(self):
        with self.assertRaises(ValueError):
            Person("Gr@ce", "Smith")

    def test_invalid_last_name(self):
        with self.assertRaises(ValueError):
            Person("Bob", "Sm1th")

    def test_str_output(self):
        p = Person("Bob", "Smith")
        self.assertEqual(str(p), "Bob,Smith")


class TestEmployee(unittest.TestCase):
    def test_default_values(self):
        e = Employee()
        self.assertEqual(e.first_name, "")
        self.assertEqual(e.last_name, "")
        self.assertEqual(e.review_date, "1900-01-01")
        self.assertEqual(e.review_rating, 3)

    def test_valid_data(self):
        e = Employee("Charlie", "Brown", "2024-10-15", 5)
```

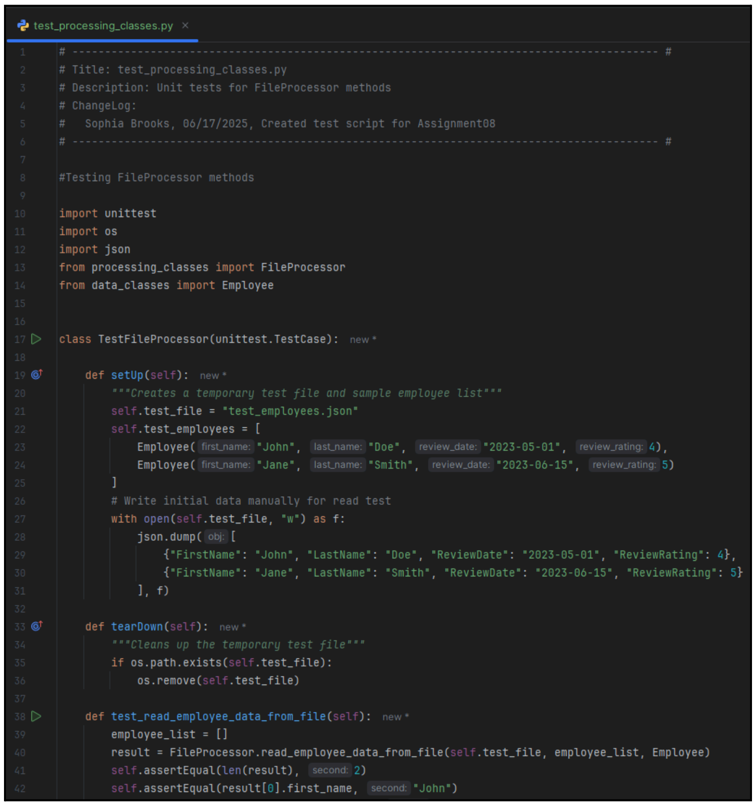
Figure 7.1


test_data_classes.py

```
class TestEmployee(unittest.TestCase):  new *
        e = Employee( first_name: "Charlie",  last_name: "Brown",  review_date: "2024-10-15",  review_rating: 5)
        self.assertEqual(e.review_date,  second: "2024-10-15")
        self.assertEqual(e.review_rating,  second: 5)

    def test_invalid_review_date(self):  new *
        with self.assertRaises(ValueError):
            Employee( first_name: "Dana",  last_name: "White",  review_date: "15-10-2024",  review_rating: 4)

    def test_invalid_review_rating(self):  new *
        with self.assertRaises(ValueError):
            Employee( first_name: "Eve",  last_name: "Black",  review_date: "2024-10-15",  review_rating: 6)

    def test_str_output(self):  new *
        e = Employee( first_name: "Alice",  last_name: "Green",  review_date: "2024-12-01",  review_rating: 4)
        self.assertEqual(str(e),  second: "Alice,Green,2024-12-01,4")


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

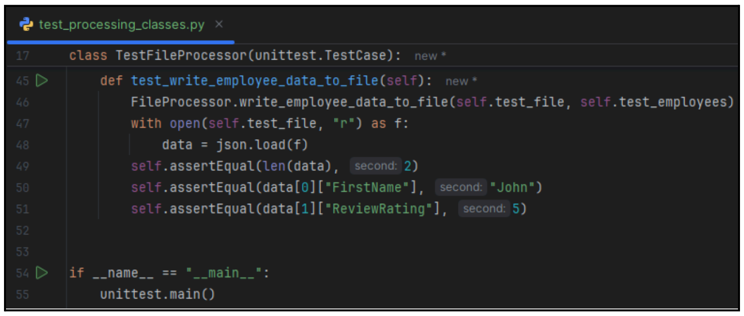
Figure 7.2

Figures 7.1 & 7.2: Showing the module script for *test_data_classes.py*.


test_processing_classes.py

```
# ------------------------------------------------------------------------------------------ #
# Title: test_processing_classes.py
# Description: Unit tests for FileProcessor methods
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created test script for Assignment08
# ------------------------------------------------------------------------------------------ #

#Testing FileProcessor methods

import unittest
import os
import json
from processing_classes import FileProcessor
from data_classes import Employee


class TestFileProcessor(unittest.TestCase):  new *

    def setUp(self):  new *
        """Creates a temporary test file and sample employee list"""
        self.test_file = "test_employees.json"
        self.test_employees = [
            Employee( first_name: "John",  last_name: "Doe",  review_date: "2023-05-01",  review_rating: 4),
            Employee( first_name: "Jane",  last_name: "Smith",  review_date: "2023-06-15",  review_rating: 5)
        ]
        # Write initial data manually for read test
        with open(self.test_file, "w") as f:
            json.dump( obj: [
                {"FirstName": "John", "LastName": "Doe", "ReviewDate": "2023-05-01", "ReviewRating": 4},
                {"FirstName": "Jane", "LastName": "Smith", "ReviewDate": "2023-06-15", "ReviewRating": 5}
            ], f)

    def tearDown(self):  new *
        """Cleans up the temporary test file"""
        if os.path.exists(self.test_file):
            os.remove(self.test_file)

    def test_read_employee_data_from_file(self):  new *
        employee_list = []
        result = FileProcessor.read_employee_data_from_file(self.test_file, employee_list, Employee)
        self.assertEqual(len(result),  second: 2)
        self.assertEqual(result[0].first_name,  second: "John")
```

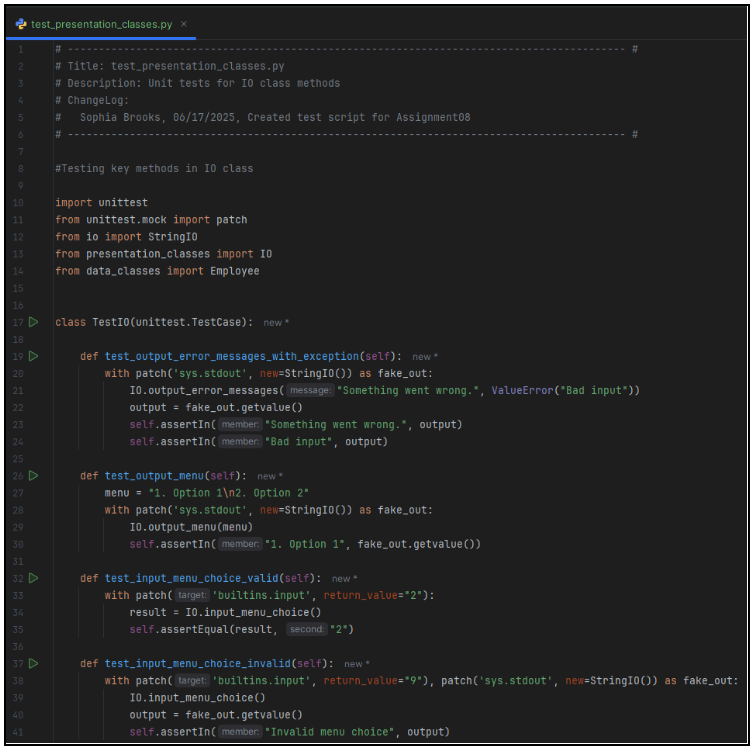
Figure 8.1


test_processing_classes.py

```
class TestFileProcessor(unittest.TestCase):  new *
    def test_write_employee_data_to_file(self):  new *
        FileProcessor.write_employee_data_to_file(self.test_file, self.test_employees)
        with open(self.test_file, "r") as f:
            data = json.load(f)
        self.assertEqual(len(data),  second: 2)
        self.assertEqual(data[0]["FirstName"],  second: "John")
        self.assertEqual(data[1]["ReviewRating"],  second: 5)


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

Figure 8.2

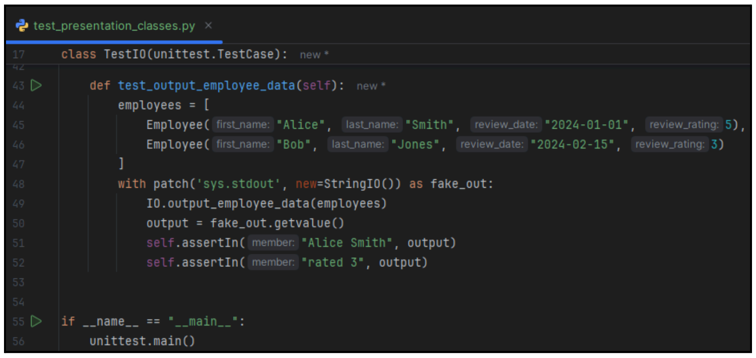
Figures 8.1 & 8.2: Showing the module script for *testing_processing_classes.py*.



test_presentation_classes.py

```python
# ------------------------------------------------------------------------------------------ #
# Title: test_presentation_classes.py
# Description: Unit tests for IO class methods
# ChangeLog:
#   Sophia Brooks, 06/17/2025, Created test script for Assignment08
# ------------------------------------------------------------------------------------------ #

#Testing key methods in IO class

import unittest
from unittest.mock import patch
from io import StringIO
from presentation_classes import IO
from data_classes import Employee


class TestIO(unittest.TestCase):

    def test_output_error_messages_with_exception(self):
        with patch('sys.stdout', new=StringIO()) as fake_out:
            IO.output_error_messages("Something went wrong.", ValueError("Bad input"))
            output = fake_out.getvalue()
            self.assertIn("Something went wrong.", output)
            self.assertIn("Bad input", output)

    def test_output_menu(self):
        menu = "1. Option 1\n2. Option 2"
        with patch('sys.stdout', new=StringIO()) as fake_out:
            IO.output_menu(menu)
            self.assertIn("1. Option 1", fake_out.getvalue())

    def test_input_menu_choice_valid(self):
        with patch('builtins.input', return_value="2"):
            result = IO.input_menu_choice()
            self.assertEqual(result, "2")

    def test_input_menu_choice_invalid(self):
        with patch('builtins.input', return_value="9"), patch('sys.stdout', new=StringIO()) as fake_out:
            IO.input_menu_choice()
            output = fake_out.getvalue()
            self.assertIn("Invalid menu choice", output)
```

Figure 9.1

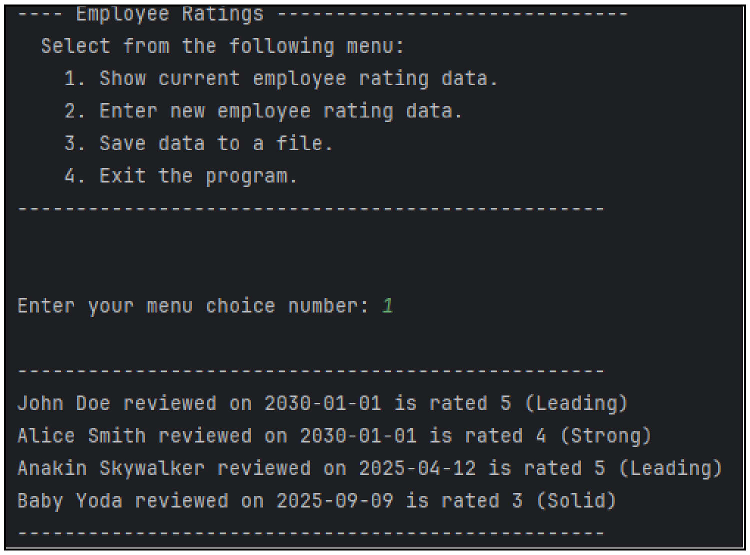


test_presentation_classes.py

```python
class TestIO(unittest.TestCase):

    def test_output_employee_data(self):
        employees = [
            Employee("Alice", "Smith", "2024-01-01", 5),
            Employee("Bob", "Jones", "2024-02-15", 3)
        ]
        with patch('sys.stdout', new=StringIO()) as fake_out:
            IO.output_employee_data(employees)
            output = fake_out.getvalue()
            self.assertIn("Alice Smith", output)
            self.assertIn("rated 3", output)


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

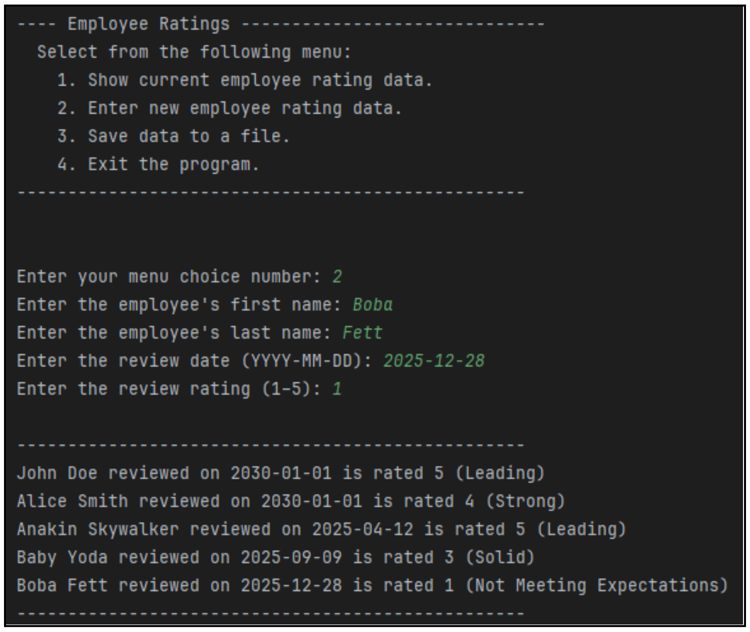
Figure 9.2

Figures 9.1 & 9.2: Showing the module script for *test_presentation_classes .py*.



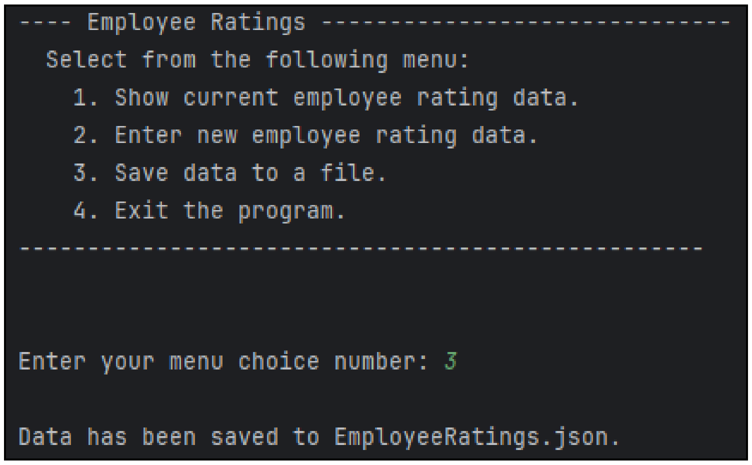
```
---- Employee Ratings ------------------------------
  Select from the following menu:
    1. Show current employee rating data.
    2. Enter new employee rating data.
    3. Save data to a file.
    4. Exit the program.
--------------------------------------------------



Enter your menu choice number: 1

--------------------------------------------------
John Doe reviewed on 2030-01-01 is rated 5 (Leading)
Alice Smith reviewed on 2030-01-01 is rated 4 (Strong)
Anakin Skywalker reviewed on 2025-04-12 is rated 5 (Leading)
Baby Yoda reviewed on 2025-09-09 is rated 3 (Solid)
--------------------------------------------------
```

Figure 10.1



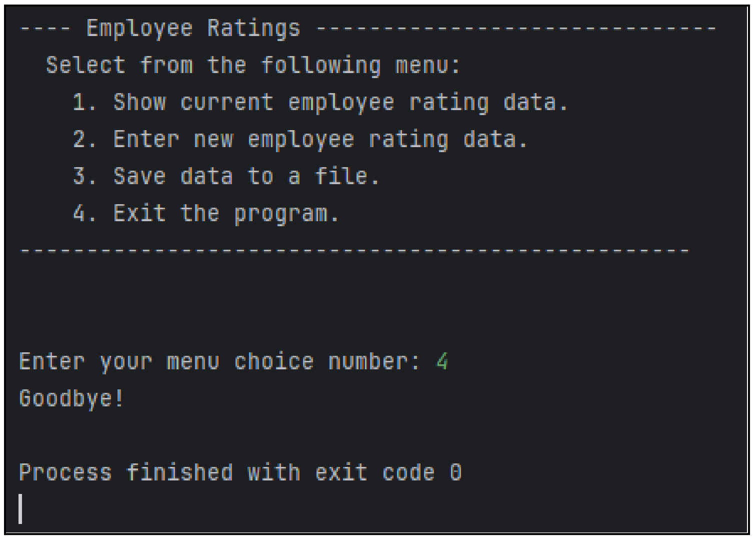
```
---- Employee Ratings ------------------------------
  Select from the following menu:
    1. Show current employee rating data.
    2. Enter new employee rating data.
    3. Save data to a file.
    4. Exit the program.
--------------------------------------------------



Enter your menu choice number: 2
Enter the employee's first name: Boba
Enter the employee's last name: Fett
Enter the review date (YYYY-MM-DD): 2025-12-28
Enter the review rating (1-5): 1

--------------------------------------------------
John Doe reviewed on 2030-01-01 is rated 5 (Leading)
Alice Smith reviewed on 2030-01-01 is rated 4 (Strong)
Anakin Skywalker reviewed on 2025-04-12 is rated 5 (Leading)
Baby Yoda reviewed on 2025-09-09 is rated 3 (Solid)
Boba Fett reviewed on 2025-12-28 is rated 1 (Not Meeting Expectations)
--------------------------------------------------
```

Figure 10.2



```
---- Employee Ratings ------------------------------
  Select from the following menu:
    1. Show current employee rating data.
    2. Enter new employee rating data.
    3. Save data to a file.
    4. Exit the program.
--------------------------------------------------



Enter your menu choice number: 3

Data has been saved to EmployeeRatings.json.
```

Figure 10.3



```
---- Employee Ratings ------------------------------
  Select from the following menu:
    1. Show current employee rating data.
    2. Enter new employee rating data.
    3. Save data to a file.
    4. Exit the program.
--------------------------------------------------



Enter your menu choice number: 4
Goodbye!

Process finished with exit code 0
```

Figure 10.4

Figures 10.1, 10.2, 10.3, & 10.4: Showing the running and correctly behaving application.

Summary

With the end of this module and course, the importance of writing well-structured, reusable, and error-resistant code becomes clear. Through hands-on labs and demonstrations, learners practice converting single-file programs into multi-module applications using the Separation of Concerns design pattern. Each class and function is moved into a logical module so that responsibilities are clearly defined and dependencies are minimized. The use of *import* statements and aliases reinforces clean connections between modules, keeping code readable and modular. Tools like UML diagrams help visualize class relationships and architectural decisions, revealing flaws such as tight coupling that can reduce flexibility. Testing techniques such as unit testing and assertions ensure that each module behaves correctly in isolation before integrating with the full application. The addition of *__name__ == "__main__"* safeguards how modules are used, ensuring users start the application from the right place and helping maintain user trust. These concepts support the development of applications that are not only functional but also resilient to change. Whether adding new features or debugging errors, a well-organized codebase makes the work significantly more manageable. Module 08 marks a turning point from writing simple scripts to engineering thoughtful, professional-grade software.