Name: Lorena Sainz-Maza Lecanda
Date: 12/08/2024
Course: Introduction to Programming with Python

Assignment 08 – Employee Review Application

Introduction

My Employee Review application is a Python-based application that helps users manage and track their employees' ratings. With features such as adding, displaying, and saving employee rating reviews into a file, it simplifies employee data management in a user-friendly interface.

To write this program, I leverage modules and imports to organize and simplify the main script (`main.py`) and add `unit test` to automatically test specific pieces of code in anticipation for any potential user errors. In this essay, I review the steps I took to create my modules and connect the modules together with the `import` command to run the program successfully. I also review my unit tests and their functionality.

Steps & Observations

1. Modules & `Import`

For this assignment, rather than having all my code in one single module, I split it into 4 different modules:

1. First, I created my `data_classes` module, where I included code for `Person` and `Employee` classes. This module does not depend on other modules in my code and, therefore, did not require any `import` dependencies, except for the `datetime` built-in library to ensure I could add error handling for the review date format.
    a. I also included my 2 constants (FILE_NAME, MENU) in this module as opposed to keeping them in main.py.
    b. All 4 attributes (`first_name, last_name, review_date and review_rating`) have getter (`@property`) and setter methods.
2. Second, I added the `presentation_classes` module, where I included all my input and output data functions. In this case, I found a dependency between this module and my data module, I added import `data_classes as data` at the very top of the module. As a result of the dependency between presentation and data, I had to make some modifications to my code.
    a. In order to output the menu successfully, I had to add `data.MENU` to the `output_menu` function.

```python
    @staticmethod  1 usage
    def output_menu(menu:str):
        """
        Displays a menu of choices to the user.

        Args:
            menu (str): A string representing the menu to be displayed. This could be a list of options
                        or a formatted string to show to the user.

        Returns:
            None: This function does not return any value, but it prints the menu to the console.
        """
        print()  # Adding extra space to make it look nicer.
        print(data.MENU)
        print()  # Adding extra space to make it look nicer.
```

b. Similarly, in order to instantiate a new employee object, I had to add `data.Employee()` to the `input_employee_data` function.

```python
        try:
            employee_object = data.Employee() # Instantiate a new employee object
            employee_object.first_name = input("What is the employee's first name? ")
            employee_object.last_name = input("What is the employee's last name? ")
            employee_object.review_date = input("What is the date of the review? ")
            employee_object.review_rating = int(input("What is the employee's rating? "))
            employee_data.append(employee_object)  # Add the employee object to the list

            print()
            print(f"You have rated {employee_object.first_name} {employee_object.last_name} "
                  f"with a rating of {employee_object.review_rating} on {employee_object.review_date}.")

        except ValueError as e:
            IO.output_error_messages( message: "That value is not the correct type of data!", e)
        except Exception as e:  # catch-all
            IO.output_error_messages( message: "There was a non-specific error!", e)
```

One last observation in this piece of code is the `output_employee_data`. Here, unlike in previous assignments, I printed specialized messages for each `employee.review_rating`.

```
    message: str = ""

    print("-" * 50)
    for employee in employee_data:
        if employee.review_rating == 5:
            message = " {} {} has been rated as 5 (GE -- Greatly Exceeds Expectations)"
        elif employee.review_rating == 4:
            message = " {} {} has been rated as 4 (EE -- Exceeds Expectations)"
        elif employee.review_rating == 3:
            message = " {} {} has been rated as 3 (ME -- Meets Expectations)"
        elif employee.review_rating == 2:
            message = " {} {} has been rated as 2 (Meets Some Expectations)"
        elif employee.review_rating == 1:
            message = " {} {} has been rated as 1 (Does Not Meet Expectations)"

        print(message.format( *args: employee.first_name, employee.last_name, employee.review_date, employee.review_rating))
    print("-" * 50)
```

3. The third module took care of reading from or writing to files. In order to do this, I had to do the following imports:
    a. `import json`: this was necessary in order to work with `json` files and use the built-in functions `json.load` for reading and `json.dump` from writing into the file.
    b. `from data_classes import Employee`: I also imported the Employee in order to be able to process the actual employee data. t represents the employee's data, such as `first_name`, `last_name`, `review_date`, and `review_rating`
    c. `from presentation_classes import IO`: This one was used for outputting error messages to the user.
4. Lastly, the `main.py` module holds the code to run the Employee Review program. Here, I imported all 3 modules:
    a. `import processing_classes as proc`: This module helps handle tasks such as reading from and writing to files.
    b. `import presentation_classes as pres`: This module involves user interface-related tasks (e.g., printing menus, receiving inputs, displaying data).
    c. `import data_classes as data`: This module helps organize constants and configure attributes (first_name, last_name, review_date and review_rating).

    In the main program code, line 16 uses the `FileProcessor` class to read employee data from a file.

```
employees = proc.FileProcessor.read_employee_data_from_file(file_name=data.FILE_NAME, employee_data=employees)
```

Next, the `while True` loop continuously presents the user with the menu until the user chooses option 4 to break out of the loop. Depending on the user's choice, the program can perform different actions:

- If the user chooses option "1", the program outputs the employee data using the `output_employee_data()` method from the `presentation_classes`. If any error exception occurs, the code triggers `output_error_messages()`.

```
24      if menu_choice == "1":
25          try:
26              pres.IO.output_employee_data(employee_data=employees)
27          except Exception as e:
28              pres.IO.output_error_messages(e)
29          continue
```

- If the user chooses option "2", the program will prompt for input using the `input_employee_data()` method to add new employee information.

```
31      elif menu_choice == "2":
32          try:
33              pres.IO.input_employee_data(employee_data=employees)
34          except Exception as e:
35              pres.IO.output_error_messages(e)
36          continue
```

- If the user chooses option "3", the program will write the updated employee data back to the file using the `write_employee_data_to_file()` method.

```
38      elif menu_choice == "3":
39          try:
40              proc.FileProcessor.write_employee_data_to_file(file_name=data.FILE_NAME, employee_data=employees)
41          except Exception as e:
42              pres.IO.output_error_messages(e)
43          continue
```

2. Unit tests

I wrote 3 unit test modules: `test_data_classes.py`, `test_presentation_classes.py` and `test_processing_classes.py`.

2.1. test_data_classes.py

This is a collection of tests for the data classes `Person` and `Employee`. The first step in the code is to import the `unittest` library as well as `Person` and `Employee` from my `data_classes` module.

```
8       import unittest
9       from data_classes import Person, Employee
```

For `Person`, I added 3 tests:

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

    def test_person_init(self):  # Tests the constructor
        person = Person( first_name: "Lucas",  last_name: "Hollis")
        self.assertEqual(person.first_name,  second: "Lucas")
        self.assertEqual(person.last_name,  second: "Hollis")

    def test_person_invalid_name(self):  # Test the first and last name validations
        with self.assertRaises(ValueError):
            person = Person( first_name: "123",  last_name: "Hollis")
        with self.assertRaises(ValueError):
            person = Person( first_name: "Lucas",  last_name: "123")

    def test_person_str(self):  # Tests the __str__() magic method
        person = Person( first_name: "Lucas",  last_name: "Hollis")
        self.assertEqual(str(person),  second: "Lucas,Hollis")
```

1. The first test validates the constructor with first name and last name.
2. The second test adds validation to make sure the first and last name have the correct value (they are alphabetic or an empty string). The `assertRaises` method is used to assert that a `ValueError` exception is raised if the user doesn't enter alphabetic characters.
3. The last test validates the `__str__` magic method. `self.assertEqual(str(person), "Lucas,Hollis")` checks if the string returned by `str(person)` is equal to the string "Lucas,Hollis".

For my `Employee` class, I added date and rating validation tests, as well as one test to check the constructor and another one to verify the correct behavior of the `__str__()` method in the `Employee` class.

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

    def test_employee_init(self):  # Tests the constructor
        employee = Employee( first_name: "Lucas",  last_name: "Hollis",  review_date: "2024-12-02",  review_rating: 4)
        self.assertEqual(employee.first_name,  second: "Lucas")
        self.assertEqual(employee.last_name,  second: "Hollis")
        self.assertEqual(employee.review_date,  second: "2024-12-02")
        self.assertEqual(employee.review_rating,  second: 4)

    def test_employee_review_date_type(self):  # Test the date validation
        with self.assertRaises(ValueError):
            employee = Employee( first_name: "Lucas",  last_name: "Hollis",  review_date: "invalid_date")

    def test_employee_review_rating_type(self):  # Test the rating validation
        with self.assertRaises(ValueError):
            employee = Employee( first_name: "Lucas",  last_name: "Hollis",  review_date: "2024-12-02",  review_rating: 0)

    def test_employee_str(self):
        student = Employee( first_name: "Lucas",  last_name: "Hollis",  review_date: "2024-12-02",  review_rating: 5)  # Te
        self.assertEqual(str(student),  second: "Lucas,Hollis,2024-12-02,5")
```

To validate `review_date` and `review_rating`, I use `assertRaises` again; for the former, this method will check if the date format is correct and, for the latter, it will make sure the rating value is not a number outside of 1-5.

```
Ran 4 tests in 0.005s

OK

Process finished with exit code 0
```

2.2. test_presentation_classes.py

First, I made 3 imports: `unittest`, `patch` from `unittest.mock` and `IO` from my `presentation_classes` module. Then, I use the `setUp` method. Here, I initialize `self.employee_data` as an empty list, which is used in the `test_input_employee_data` test method.

```
8   import unittest
9   from unittest.mock import patch
10  from presentation_classes import IO
11
12  class TestIO(unittest.TestCase):
13      def setUp(self):
14          self.employee_data = []
```

The first test, `test_input_menu_choice`, checks if the `input_menu_choice` method correctly processes user input. I use `patch` to mock the input function and simulate user input, specifically the string '2' for the menu choice.

```
16      def test_input_menu_choice(self):
17          # Simulate user input '2' and check if the function returns '2'
18          with patch( target: 'builtins.input', return_value='2'):
19              choice = IO.input_menu_choice()
20              self.assertEqual(choice,  second: '2')
21
```

```
Ran 1 test in 0.003s

OK

Process finished with exit code 0
```

The second test simulates the user input for entering employee data and verifies if it is correctly added to the `employee_data` list. The test checks 3 different scenarios: one where the user enter valid input data and two scenarios where the input contains invalid date format and invalid rating (non-integer).

```python
def test_input_employee_data(self):
    # Simulate user input for employee data
    with patch( target: 'builtins.input', side_effect=['Lucas', 'Hollis', '2024-12-03', '5']):
        IO.input_employee_data(self.employee_data)
        self.assertEqual(len(self.employee_data), second: 1)
        self.assertEqual(self.employee_data[0].first_name, second: 'Lucas')
        self.assertEqual(self.employee_data[0].last_name, second: 'Hollis')
        self.assertEqual(self.employee_data[0].review_date, second: '2024-12-03')
        self.assertEqual(self.employee_data[0].review_rating, second: 5)

    # Simulate invalid review date input (incorrect format)
    with patch( target: 'builtins.input', side_effect=['Lucas', 'Hollis', 'invalid', '5']):
        IO.input_employee_data(self.employee_data)
        self.assertEqual(len(self.employee_data), second: 1)  # Data should not be added due to invalid input

    # Simulate invalid review rating input (non-integer)
    with patch( target: 'builtins.input', side_effect=['Lucas', 'Hollis', '2024-12-03', 'invalid']):
        IO.input_employee_data(self.employee_data)
        self.assertEqual(len(self.employee_data), second: 1)  # Data should not be added due to invalid input
```

In this piece of code, I'm still not 100% sure what the purpose of adding length of 1 is or if this is strictly required. I mainly just added this part because I saw it in the Lab Answer code; I did run the code in chatGPT to get a better understanding of the meaning and I believe it means that the user will not be able to add new input if the input for date and rating are invalid. The test seems to run successfully.

```
You have rated Lucas Hollis with a rating of 5 on 2024-12-03.
That value is not the correct type of data!

-- Technical Error Message --
Incorrect data format, should be YYYY-MM-DD
Inappropriate argument value (of correct type).
<class 'ValueError'>
That value is not the correct type of data!

-- Technical Error Message --
Please choose only values 1 through 5
Inappropriate argument value (of correct type).
<class 'ValueError'>


Ran 1 test in 0.004s

OK
```

2.3. test_processing_classes.py

For my last collection of tests, I validated the functions within my `FileProcessor` class in the `processing_classes` module. In addition to importing several libraries and my modules, I used the `setUp` method to create and delete a temporary file for testing.

```python
import unittest
import tempfile
import json
import data_classes as data
from processing_classes import FileProcessor


class TestFileProcessor(unittest.TestCase):
    def setUp(self):
        # Create a temporary file for testing
        self.temp_file = tempfile.NamedTemporaryFile(delete=False)
        self.temp_file_name = self.temp_file.name
        self.employee_data = []

    def tearDown(self):
        # Clean up and delete the temporary file
        self.temp_file.close()
```

In order to test the `test_read_employee_data_from_file` function, I followed these steps:

1. I created some sample data and wrote it into a temporary file
2. I called the read_employee_data_from_file method to check if it returns the expected data
3. I validate that the employee data list contains the expected employee objects

```python
    def test_read_employee_data_from_file(self):
        # Create some sample data and write it to the temporary file
        sample_data = [
            {"FirstName": "Lucas", "LastName": "Hollis", "ReviewDate": "2024-12-04", "ReviewRating": 5},
            {"FirstName": "Mike", "LastName": "Hollis", "ReviewDate": "2024-12-04", "ReviewRating": 3},
            {"FirstName": "Lorena", "LastName": "Hollis", "ReviewDate": "2024-12-04", "ReviewRating": 4}
        ]
        with open(self.temp_file_name, "w") as file:
            json.dump(sample_data, file)

        # Call the read_data_from_file method and check if it returns the expected data
        FileProcessor.read_employee_data_from_file(self.temp_file_name, self.employee_data)

        # Assert that the employee_data list contains the expected employee objects
        self.assertEqual(len(self.employee_data), len(sample_data))
        self.assertEqual(self.employee_data[0].last_name, second: "Hollis") # Check one attribute of the f
        self.assertEqual(self.employee_data[1].review_date, second: "2024-12-04") # Check one attribute of
        self.assertEqual(self.employee_data[2].first_name, second: "Lorena") # Check one attribute of the
        self.assertEqual(self.employee_data[2].review_rating, second: 4) # Check one attribute of the thir
```

```
Ran 1 test in 0.007s

OK

Process finished with exit code 0
```

For the `test_write_employee_data_to_file`:

1. I created some sample employee objects
2. I called the `write_employee_data_to_file` method to write the data into the temporary file
3. I read the data from the temporary file and check if it matches the expected JSON data

```python
def test_write_employee_data_to_file(self):
    # Create some sample employee objects
    sample_employees = [
        data.Employee( first_name: "Lucas",  last_name: "Hollis",  review_date: "2024-12-04",  review_rating: 5),
        data.Employee( first_name: "Mike",  last_name: "Hollis",  review_date: "2024-12-04",  review_rating: 3),
        data.Employee( first_name: "Lorena",  last_name: "Hollis",  review_date: "2024-12-04",  review_rating: 4)
    ]

    # Call the write_data_to_file method to write the data to the temporary file
    FileProcessor.write_employee_data_to_file(self.temp_file_name, sample_employees)

    # Read the data from the temporary file and check if it matches the expected JSON data
    with open(self.temp_file_name, "r") as file:
        file_data = json.load(file)

    self.assertEqual(len(file_data), len(sample_employees))
    self.assertEqual(file_data[0]["FirstName"],  second: "Lucas")
    self.assertEqual(file_data[1]["ReviewDate"],  second: "2024-12-04")
```

```
Ran 1 test in 0.007s

OK

Process finished with exit code 0
```

Running the main script

With this program, I am able to successfully to iterate over the following tasks (until exiting the program):

1. input for an employee's first, last name, review date and rating

```
Enter your menu choice number: 2
What is the employee's first name? Lucas
What is the employee's last name? Hollis
What is the date of the review? 2024-12-08
What is the employee's rating? 5

You have rated Lucas Hollis with a rating of 5 on 2024-12-08.
```

2. display my input for employee's first, last name, review date and rating.

```
Enter your menu choice number: 1
-------------------------------------------------
 Lucas Hollis has been rated as 5 (GE -- Greatly Exceeds Expectations)
 Mike Hollis has been rated as 5 (GE -- Greatly Exceeds Expectations)
 Lorena Hollis has been rated as 5 (GE -- Greatly Exceeds Expectations)
-------------------------------------------------
```

3. save the data into "EmployeeRatings.json" as valid JSON syntax

```json
[
    {
        "FirstName": "Lucas",
        "LastName": "Hollis",
        "ReviewDate": "2024-12-08",
        "ReviewRating": 5
    },
    {
        "FirstName": "Mike",
        "LastName": "Hollis",
        "ReviewDate": "2024-12-08",
        "ReviewRating": 5
    },
    {
        "FirstName": "Lorena",
        "LastName": "Hollis",
        "ReviewDate": "2024-12-08",
        "ReviewRating": 5
    }
]
```

Conclusion

I completed the last assignment in multiple days. I tried to not use the Assignment08starter and instead wrote the code from scratch using my own code from Assignment07. I chose to first write the entire code in the same python file and then moved each class to a different module, leaving the code to run the program in the main.py file. To connect each module to each other, I tried to use my own intuition and understanding first and, later, I checked the Lab03 modules since that Lab was very similar to this assignment. Lastly, to write my unit test, I also used the files available in Mod08-Lab03. I would not have been able to figure out how to write the syntax to run these tests without these additional resources. I reflect more on this in Task 5.