Walden Marcus

03/23/2025

IT FDN 110 A

Assignment 08

Assignment 08

Link to Github

Introduction

Assignment08 is a Python program designed to receive input regarding employee ratings and store them to a JSON file. The program reads from a designated JSON file prior to input, and writes the total prior and current input back to the same file through program use. Assignment08 includes unit testing for the three types of classes involved in the program. Please enjoy the inclusion of the cast and characters from movie sensation, “Dr. Horrible’s Sing-Along Blog” (2008) for the purposes of unit testing and sample data.

Data Classes

There are two classes in this section: Person, and Employee(Person). The two functions associated with Person are Person(first_name) and Person(last_name), which respectively hold and return the first and last names of individuals entered into the program’s inquiries. Both of these functions include setters that check for alphabetical versus numeric/character input. Lastly, the __str__() value is overridden for the purposes of this program’s use cases.

```
from datetime import date
class Person:
    def __init__(self, first_name: str = "", last_name: str = ""):
        self.first_name = first_name
        self.last_name = last_name
    @property
    def first_name(self):
        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.")
    @property
    def last_name(self):
        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.")
    def __str__(self):
        return f"{self.first_name},{self.last_name}"
```

Fig 1. *Notice how date is imported from the package datetime. This will be relevant shortly.*

The two functions associated with Employee(Person), a child class of Person, are Employee.review_date and Employee.review_rating. Employee.review_date holds and returns the date of an employee's review, and Employee.review_rating holds the integer between 1-5 that is their rating. This data is validated by checking against a function of datettime for date formatting, and against a list of acceptable rating values.

```
class Employee(Person):
    def __init__(self, first_name: str = "", last_name: str = "", review_date: str = "1900-01-01",
review_rating: int = 3):
        super().__init__(first_name=first_name,last_name=last_name)
        self.review_date = review_date
        self.review_rating = review_rating
    @property
    def review_date(self):
        return self.__review_date
    @review_date.setter
    def review_date(self, value: str):
        try:
            date.fromisoformat(value)
            self.__review_date = value
        except ValueError:
            raise ValueError("Incorrect data format, should be YYYY-MM-DD")
    @property
```

```
    def review_rating(self):
        return self.__review_rating
    @review_rating.setter
    def review_rating(self, value: str):
        if value in (1, 2, 3, 4, 5):
            self.__review_rating = value
        else:
            raise ValueError("Please choose only values 1 through 5")
    def __str__(self):
        return f"{self.first_name},{self.last_name},{self.review_date},{self.__review_rating}"
```

Fig 2. *Notice in the __init__ of Employee(Person), the class is indicated as the child class of Person.*

Unit Testing

The unit testing for data_classes tests the following: 1) Whether the Person.__init__ recognizes input, 2) Whether Person.first_name and Person.last_name correctly raise exceptions in the case of numeric input, 3) Whether Person.__str__ processes as intended, 4) Whether Employee.__init__ recognizes input, 4) Whether Employee.review_date and Employee.review_rating correctly raise exceptions in the case of invalid input, and 5) Whether Employee.__str__ processes as intended.

```
    import unittest
    from data_classes import Person, Employee
    class TestPerson(unittest.TestCase):
        """
        A class to test the constructor, first/last name validation, and __str__() of Person
        Changelog:
        Wmarcus, 3/23/25, Created class
        """
        def test_person_init(self):  # Tests the constructor
            person = Person("Johnny", "Snow")
            self.assertEqual(person.first_name, "Johnny")
            self.assertEqual(person.last_name, "Snow")
        def test_person_invalid_name(self):  # Test the first and last name validations
            with self.assertRaises(ValueError):
                person = Person("123", "Snow")
            with self.assertRaises(ValueError):
                person = Person("Johnny", "123")
```

```
    def test_person_str(self):  # Tests the __str__() magic method
        person = Person("Johnny", "Snow")
        self.assertEqual(str(person), "Johnny,Snow")
class TestEmployee(unittest.TestCase):
    def test_employee_init(self):  # Tests the constructor
        employee = Employee("Bad", "Horse", "2008-07-15", 1)
        self.assertEqual(employee.first_name, "Bad")
        self.assertEqual(employee.last_name, "Horse")
        self.assertEqual(employee.review_date, "2008-07-15")
        self.assertEqual(employee.review_rating, 1)
    def test_employee_invalid_review_date(self):
        with self.assertRaises(ValueError):
            student = Employee("Neil", "Harris", "2008-7-15", 5)
    def test_employee_invalid_review_rating(self):
        with self.assertRaises(ValueError):
            student = Employee("Nathan", "Fillion", "2008-07-15", "five")
    def test_employee_str(self):
        employee = Employee("Felicia", "Day", "2008-07-15", 5)
        self.assertEqual(str(employee), "Felicia,Day,2008-07-15,5")
if __name__ == '__main__':
    unittest.main()
```

Fig 3. *Notice how unittest, a package created for this very purpose, is imported in order to streamline unit testing. Additionally, Person and Employee are imported for recognition.*

Presentation Classes

Within the single class IO, there are five functions. The function output_error_messages() displays technical error messages when an exception is raised. The function output_menu() displays the menu string defined later in Main Logic. The function input_menu_choice() receives and returns user input pertaining to options displayed from output_menu(). The function output_employee_data() displays data previously read from the JSON file (then saved to a variable) and any data entered during the session of program use. In particular, this function attributes certain messages pertaining to employee rating. The function input_employee_data assigns user input to attributes of the object employee_object, then appends the data to the list employee_data. ValueError and Exception may be raised depending on user input errors.

```
class IO:
    @staticmethod
```

```
    def output_error_messages(message: str, error: Exception = None):
        print(message, end="\n\n")
        if error is not None:
            print("-- Technical Error Message -- ")
            print(error, error.__doc__, type(error), sep='\n')
    @staticmethod
    def output_menu(menu: str):
        print()
        print(menu)
        print()
    @staticmethod
    def input_menu_choice():
        choice = "0"
        try:
            choice = input("Enter your menu choice number: ")
            if choice not in ("1", "2", "3", "4"):  # Note these are strings
                raise Exception("Please, choose only 1, 2, 3, or 4")
        except Exception as e:
            IO.output_error_messages(e.__str__())
        return choice
    @staticmethod
    def output_employee_data(employee_data: list):
        message:str = ''
        print()
        print("-" * 50)
        for employee in employee_data:
            if employee.review_rating == 5:
                message = " {} {} is rated as 5 (Leading)"
            elif employee.review_rating == 4:
                message = " {} {} is rated as 4 (Strong)"
            elif employee.review_rating == 3:
                message = " {} {} is rated as 3 (Solid)"
            elif employee.review_rating == 2:
                message = " {} {} is rated as 2 (Building)"
            elif employee.review_rating == 1:
                message = " {} {} is rated as 1 (Not Meeting Expectations)"
            print(message.format(employee.first_name, employee.last_name, employee.review_date,
employee.review_rating))
```

```
        print("-" * 50)
        print()
    @staticmethod
    def input_employee_data(employee_data: list, employee_type: Employee):
        try:
            employee_object = employee_type()
            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 their review date? ")
            employee_object.review_rating = int(input("What is their review rating? "))
            employee_data.append(employee_object)
        except ValueError as e:
            IO.output_error_messages("That value is not the correct type of data!", e)
        except Exception as e:
            IO.output_error_messages("There was a non-specific error!", e)
        return employee_data
```

Fig 4. *The methods associated with presentation_classes.*

Unit Testing

Unit testing for the Presentation Classes includes use of the unittest and unittest.mock packages. Additionally, IO and Employee are imported from their respective modules. Unit testing for these classes tests the following: 1) Whether input_menu_choice recognizes input, 2) Whether input_employee_data recognizes input, and 3) Whether invalid data input results in data addition or not.

```
import unittest
from unittest.mock import patch
from presentation_classes import IO
from data_classes import Employee
class TestIO(unittest.TestCase):
    def setUp(self):
        self.employee_data = []
    def test_input_menu_choice(self):
        # Simulate user input '2' and check if the function returns '2'
        with patch('builtins.input', return_value='2'):
            choice = IO.input_menu_choice()
            self.assertEqual(choice, '2')
    def test_input_employee_data(self):
```

```
        # Simulate user input for employee data
        with patch('builtins.input', side_effect=['Doctor', 'Horrible', '2008-07-15', 1]):
            IO.input_employee_data(self.employee_data, employee_type=Employee)
            self.assertEqual(len(self.employee_data), 1)
            self.assertEqual(self.employee_data[0].first_name, 'Doctor')
            self.assertEqual(self.employee_data[0].last_name, 'Horrible')
            self.assertEqual(self.employee_data[0].review_date, '2008-07-15')
            self.assertEqual(self.employee_data[0].review_rating, 1)
        # Simulate invalid date input
        with patch('builtins.input', side_effect=['Captain', 'Hammer', 'invalid', 5]):
            IO.input_employee_data(self.employee_data, employee_type=Employee)
            self.assertEqual(len(self.employee_data), 1)
        # Simulate invalid rating input (not an int)
        with patch('builtins.input', side_effect=['Captain', 'Hammer', '2008-07-15', 'invalid']):
            IO.input_employee_data(self.employee_data, employee_type=Employee)
            self.assertEqual(len(self.employee_data), 1)
if __name__ == "__main__":
    unittest.main()
```

Fig 5. *The unit testing associated with the Presentation Classes.*

Processing Classes

Within the single class FileProcessor, there are two methods: read_employee_data_from_file, which reads data from “Enrollments.JSON” and adds it to the variable employee_data, and write_employee_data_to_file, which conversely converts the list of dictionaries employee_data to a format suitable to overwrite “Enrollments.JSON”.

```
from data_classes import Employee
import json
class FileProcessor:
    @staticmethod
  def read_employee_data_from_file(file_name: str, employee_data: list, employee_type: Employee):
          try:
            with open(file_name, "r") as file:
                list_of_dictionary_data = json.load(file)
                for employee in list_of_dictionary_data:
                    employee_object = employee_type()
                    employee_object.first_name = employee["FirstName"]
```

```
                employee_object.last_name = employee["LastName"]
                employee_object.review_date = employee["ReviewDate"]
                employee_object.review_rating = employee["ReviewRating"]
                employee_data.append(employee_object)
        except FileNotFoundError:
            raise FileNotFoundError("Text file must exist before running this script!")
        except Exception:
            raise Exception("There was a non-specific error!")
        return employee_data
    @staticmethod
    def write_employee_data_to_file(file_name: str, employee_data: list):
        try:
            list_of_dictionary_data: list = []
            for employee in employee_data:
                employee_json: dict = {"FirstName": employee.first_name,
                                       "LastName": employee.last_name,
                                       "ReviewDate": employee.review_date,
                                       "ReviewRating": employee.review_rating
                                       }
                list_of_dictionary_data.append(employee_json)

            with open(file_name, "w") as file:
                json.dump(list_of_dictionary_data, file)
        except TypeError:
            raise TypeError("Please check that the data is a valid JSON format")
        except PermissionError:
            raise PermissionError("Please check the data file's read/write permission")
        except Exception as e:
            raise Exception("There was a non-specific error!")
```

Fig 6. *Notice how, like most functions in this program, functions are annotated with @staticmethod and include exception handling.*

Unit Testing

The unit testing for the processing classes tests whether a temporary file with sample data is read from and written to in an expectable format. By creating temporary files and sample datasets, the unit testing can examine the functionalities of the classes without using or manipulating the actual JSON or list data.

```
import unittest

import tempfile

import json

import data_classes as data

from data_classes import Employee

from processing_classes import FileProcessor

class TestFileProcessor(unittest.TestCase):

    def setUp(self):

        self.temp_file = tempfile.NamedTemporaryFile(delete=False)

        self.temp_file_name = self.temp_file.name

        self.employee_data = []

    def tearDown(self):

        self.temp_file.close()

    def test_read_data_from_file(self):

        sample_data = [

            {"FirstName": "Neil", "LastName": "Harris", "ReviewDate":"2008-07-15",
"ReviewRating":5},

            {"FirstName": "Nathan", "LastName": "Fillion", "ReviewDate": "2008-07-15",
"ReviewRating": 5}

        ]

        with open(self.temp_file_name, "w") as file:

            json.dump(sample_data, file)

        FileProcessor.read_employee_data_from_file(self.temp_file_name, self.employee_data,
employee_type=Employee)

        self.assertEqual(len(self.employee_data), len(sample_data))

        self.assertEqual(self.employee_data[0].first_name, "Neil")

        self.assertEqual(self.employee_data[0].last_name, "Harris")

        self.assertEqual(self.employee_data[0].review_date, "2008-07-15")

        self.assertEqual(self.employee_data[0].review_rating, 5)

        self.assertEqual(self.employee_data[1].first_name, "Nathan")

        self.assertEqual(self.employee_data[1].last_name, "Fillion")

        self.assertEqual(self.employee_data[1].review_date, "2008-07-15")

        self.assertEqual(self.employee_data[1].review_rating, 5)

    def test_write_data_to_file(self):

        sample_employees = [

            data.Employee("Neil", "Harris", "2008-07-15", 5),

            data.Employee("Nathan", "Fillion", "2008-07-15", 5)

        ]

        FileProcessor.write_employee_data_to_file(self.temp_file_name, sample_employees)
```

```
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"], "Neil")
        self.assertEqual(file_data[0]["LastName"], "Harris")
        self.assertEqual(file_data[0]["ReviewDate"], "2008-07-15")
        self.assertEqual(file_data[0]["ReviewRating"], 5)
        self.assertEqual(file_data[1]["FirstName"], "Nathan")
        self.assertEqual(file_data[1]["LastName"], "Fillion")
        self.assertEqual(file_data[1]["ReviewDate"], "2008-07-15")
        self.assertEqual(file_data[1]["ReviewRating"], 5)
if __name__ == "__main__":
    unittest.main()
```

Fig 7. *Notice how the package tempfile is additionally imported to manage the temporary file/s associated with unit testing.*

Main Logic

The module main.py contains the constants, global variables, and logic for the program. It imports the presentation and processing classes under the aliases pres and proc, and imports Employee from data_classes. The file name and menu are both constants, and the list of employees and the menu choice are both returned global variables. Upon program initiation, Assignment 08 reads the contents of “Enrollments.JSON” to employee and displays MENU to the user, thus beginning the while loop of options, terminated by user selection to break the loop.

```
import presentation_classes as pres
import processing_classes as proc
from data_classes import Employee
# Data ------------------------------------------ #
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.
```

```
------------------------------------------------
'''
employees: list = []
menu_choice = ''
employee = proc.FileProcessor.read_employee_data_from_file(file_name=FILE_NAME,
                                                           employee_data=employees,
                                                           employee_type=Employee)
while True:
    pres.IO.output_menu(menu=MENU)

    menu_choice = pres.IO.input_menu_choice()

    if menu_choice == "1":
        try:
            pres.IO.output_employee_data(employee_data=employees)
        except Exception as e:
            pres.IO.output_error_messages(e)
        continue
    elif menu_choice == "2":
        try:
            employees = pres.IO.input_employee_data(employee_data=employees,
employee_type=Employee)
            pres.IO.output_employee_data(employee_data=employees)
        except Exception as e:
            pres.IO.output_error_messages(e)
        continue
    elif menu_choice == "3":
        try:
            proc.FileProcessor.write_employee_data_to_file(file_name=FILE_NAME,
employee_data=employees)
            print(f"Data was saved to the {FILE_NAME} file.")
        except Exception as e:
            pres.IO.output_error_messages(e)
        continue
    elif menu_choice == "4":
        break
```

Fig 8. The logic of Assignment 08.

Summary

Under the expected use cases of Assignment08, users should be able to read, write, and recall data associated with the JSON file. Additionally, the data validation of Assignment08 ensures that all data entered is entered in a correct format to read from the JSON file should the program be initiated after an initial use. Expansion to this program includes improvements to user experience through a windowed application.