

Ophir Amon

3/11/2024

Foundations of Programming: Python

Assignment08

GitHub link: <https://github.com/OphirAmon13/IntroToProg-Python-Mod08>

Assignment 08 Steps

Introduction

Our eighth and final assignment was similar to the previous assignment, as it dealt with loops, conditional statements, lists, dictionaries, error handling, functions, classes, JSON files, objects and inheritance. However, this assignment added two new topics: unit testing and the idea of storing different pieces of the code in separate files. The difference in this assignment was that we were required to use the Employee class instead of the Student class, as well as provide unit testing for all functions and separating code into separate files. I separated my code into four different files: one that dealt with all input/output with the user, another that dealt with any processing that was done with JSON files, one that stores all my data classes, and lastly, a file that runs my main method.

Loops

In general, loops are helpful for when you want to perform a task repeatedly. I used two different types of loops for this program: a 'while' loop and a 'for' loop. 'While' loops are helpful for when you want to iterate your program based on a condition. In this case, my condition was while the program was TRUE (Figure 1). 'For' loops are good for when we know exactly how many times we want to iterate and are therefore good when looping through lists as I did in this program (Figure 2).

```
while True:
```

Figure 1: Python 'While' Loop Example.

```
for employee in employee_data:  
    print(f"{employee.first_name} \t\t{employee.last_name} \t\t{employee.review_date} \t\t{employee.review_rating}")
```

Figure 2: Python 'For' Loop Example

Conditional Statements

Much like a 'while' loop, conditional statements only run if a criterion is met. For this program, I used the match method. This method allows me to consider however many cases I want by simply labeling what the case should be. This program had 5 base cases, based on what menu option the user inputs (Figure 3). The first option allowed the user to register an employee with a specific review date and review rating. The user just needs the first name, last name, review date, and review rating to complete the first task. The second task shows the current data that the user has input so far. This can include registration information for multiple students. The third option takes in the input data and creates a JSON file with the data, as well as informing the user that they have registered students for their courses. The fourth option ends the program by using the exit() function. The last option (case _) is for when the user selects an option that is not on the menu. When this option runs, it simply prints the message "Please select a valid option".

```
# Checks the user's menu choice against different cases
match menu_choice:
    case "1":
        # Allows user to add employee data
        presentation_classes.IO.input_employee_data(employees)
    case "2":
        # Prints the current data
        presentation_classes.IO.output_employee_data(employees)
    case "3":
        # Adds data to file
        processing_classes.FileProcessor.write_data_to_file(FILE_NAME, employees)
    case "4":
        # Exits program
        presentation_classes.IO.quit_program()
    case _:
        # Handles all other inputs
        print("ERROR: Please select a valid option")
```

Figure 3: Python Conditional Statements Examples

JSON Files

As mentioned above, the third option of choices was to create a JSON text file with the information retrieved from the user. To do this, I used the ‘open’ function and set it to the ‘w’ (writing) mode. The open() function opens a new file with any desired name and type. In this case, I opened a JSON file named ‘EmployeeRatings.json’ and began editing the contents of the file (Figure 4).

```
# Function that saves data to a file
@staticmethod
def write_data_to_file(file_name: str, employee_data: list): # Saves all information to JSON file
    try:
        list_of_employees: list = []
        for employee in employee_data:
            employee_dict: dict = {
                "first_name": employee.first_name,
                "last_name": employee.last_name,
                "review_date": employee.review_date,
                "review_rating": employee.review_rating
            }
            list_of_employees.append(employee_dict)
        with open(file_name, "w") as file:
            json.dump(list_of_employees, file)
        for employee in list_of_employees:
            print(f"You have reviewed the employee, {employee['first_name']} {employee['last_name']}, on {employee['review_date']}, with a rating of {employee['review_rating']}.")
    except Exception as error_message:
        presentation_classes.IO.output_error_messages(f"There was an error saving the data to the {file_name} file!", error_message)
```

Figure 4: Python Open() Function Example.

Lists

Lists are useful for storing multiple pieces of data into one collection. For this program I used lists so that the user can input registration information for multiple employees at the same time. The list I used in this assignment stored Employees in its contents (Figure 5).

```
employee_first_name = input("Enter the employee's first name: ")
employee_last_name = input("Enter the employee's last name: ")
review_date = input("Please enter the review date (YYYY-MM-DD): ")
review_rating = input("Please enter the review rating (1-5): ")
new_employee = data_classes.Employee(employee_first_name, employee_last_name, review_date, review_rating)
employee_data.append(new_employee) # Adds employees to list of all employee data
```

Figure 5: Python Lists Example.

Dictionaries

Dictionaries are useful for storing data in key-value pairs. For each key in a dictionary there is a corresponding value. For this assignment, the keys were the strings “first_name”, “last_name”,

and “review_date”, and “review_rating” while the values were the employee.first_name, employee.last_name, and employee.review_date, and employee.review_rating that the user inputted (Figure 6).

```
employee_dict: dict = {
    "first_name": employee.first_name,
    "last_name": employee.last_name,
    "review_date": employee.review_date,
    "review_rating": employee.review_rating
}
```

Figure 6: Python Dictionaries Example.

Error Handling

Error handling is a very important aspect for the programmer to account for. Often, while writing code, the program can encounter many errors. In doing so, the program crashes, which is not very desirable. So, to ensure that does not happen, while developing the code, one should anticipate any potential errors that might come up and “handle” them. For example, the code for this assignment had a requirement that whenever the code was run, it read data from a file called EmployeeRatings.json. However, if the computer running the program did not have such a file stored, the program would crash and a FileNotFoundError would appear. To avoid having the program crash, I used a try-except block. The computer will try to execute whatever is under the try block and if it encounters an error accounted for in the except block it will print an error message instead of crashing (Figure 7).

```
# Function that reads and prints the data from a file
@staticmethod
def read_data_from_file(file_name: str, employee_data: list):
    print(file_name)
    try:
        with open(file_name, "r") as file:
            print("First Name \tLast Name \tReview Date \tReview Rating")
            loaded_employee_table = json.load(file)
            for item in loaded_employee_table:
                new_employee = data_classes.Employee(item["first_name"], item["last_name"],
item["review_date"], item["review_rating"])
                employee_data.append(new_employee)
                print(f"{new_employee.first_name} \t\t\t{new_employee.last_name}
\t\t\t{new_employee.review_date} \t\t\t{new_employee.review_rating}")
    except FileNotFoundError as error_message:
        presentation_classes.IO.output_error_messages(f"There was an error finding the {file_name} file!",
error_message)
    except JSONDecodeError as error_message:
        presentation_classes.IO.output_error_messages(f"There was an error reading the data from the
{file_name} file!", error_message)
    except Exception as error_message:
        presentation_classes.IO.output_error_messages(f"There was an error reading the data from the
{file_name} file!", error_message)
```

Figure 7: Python Error Handling Example.

Functions

Functions are very useful as they can make code much neater. A function is a block of code that only runs when it is called. Functions are especially useful for when you want to run the same

block of code multiple times, as instead of having the same block of code for each instance, you can just call the function each time. Functions can take in parameters as well. This means that it can use variables that are not explicitly written in the function itself. I defined all my functions in their respective files and called any function I needed under the main block, making my code much nicer visually (Figure 8).

```
if __name__ == "__main__":
    # Reads any data from file
    processing_classes.FileProcessor.read_data_from_file(FILE_NAME, employees)
    while True:
        # Present the menu of choices
        presentation_classes.IO.output_menu(MENU)
        menu_choice = presentation_classes.IO.input_menu_choice()
        # Checks the user's menu choice against different cases
        match menu_choice:
            case "1":
                # Allows user to add employee data
                presentation_classes.IO.input_employee_data(employees)
            case "2":
                # Prints the current data
                presentation_classes.IO.output_employee_data(employees)
            case "3":
                # Adds data to file
                processing_classes.FileProcessor.write_data_to_file(FILE_NAME, employees)
            case "4":
                # Exits program
                presentation_classes.IO.quit_program()
            case _:
                # Handles all other inputs
                print("ERROR: Please select a valid option")
```

Figure 8: Python Functions Example.

Classes

Classes are also helpful for making code neater. For this assignment I used four classes: one called Person, one called Employee, a FileProcessor class, and an IO class. In each class, I included the functions that had to do with those categories in their respective class (Figure 9).

```
10 class Person: # Creates the Person object
11     # Initializes the Person object
12     def __init__(self, first_name: str, last_name: str):
13     def __str__(self):
14     @property
15     def first_name(self):
16     @first_name.setter
17     def first_name(self, value: str):
18     @property
19     def last_name(self):
20     @last_name.setter
21     def last_name(self, value: str):
22
23 class Employee(Person): # Creates the Employee object which inherits from the Person class
24     # Initializes the Employee object
25     def __init__(self, first_name: str = "", last_name: str = "", review_date: datetime.date = "1990-01-01", review_rating: int = 3):
26     def __str__(self):
27     def __repr__(self):
28     @property
29     def review_date(self):
30     @review_date.setter
31     def review_date(self, value):
32     @property
33     def review_rating(self):
34     @review_rating.setter
35     def review_rating(self, value):
36
37 class FileProcessor: # Stores all functions that have to do with .json files
38     # Function that reads and prints the data from a file
39     @staticmethod
40     def read_data_from_file(file_name: str, employee_data: list):
41     try:
42         with open(file_name, "r") as file:
43             print("First Name \tlast Name \tReview Date \tReview Rating")
44             loaded_employee_table = json.load(file)
45             for item in loaded_employee_table:
46                 new_employee = data_classes.Employee(item["first_name"], item["last_name"], item["review_date"], item["review_rating"])
47                 employee_data.append(new_employee)
48                 print(f"{new_employee.first_name} \t\t{new_employee.last_name} \t\t{new_employee.review_date} \t\t{new_employee.review_rating}")
49     except FileNotFoundError as error_message:
50         presentation_classes.IO.output_error_messages(f"There was an error finding the {file_name} file!", error_message)
51     except JSONDecodeError as error_message:
52         presentation_classes.IO.output_error_messages(f"There was an error reading the data from the {file_name} file!", error_message)
53     except Exception as error_message:
54         presentation_classes.IO.output_error_messages(f"There was an error reading the data from the {file_name} file!", error_message)
55
56     # Function that saves data to a file
57     @staticmethod
58     def write_data_to_file(file_name: str, employee_data: list): # Saves all information to JSON file
59     try:
60         list_of_employees: list = []
61         for employee in employee_data:
62             employee_dict: dict = {
63                 "first_name": employee.first_name,
64                 "last_name": employee.last_name,
65                 "review_date": employee.review_date,
66                 "review_rating": employee.review_rating
67             }
68             list_of_employees.append(employee_dict)
69
70         with open(file_name, "w") as file:
71             json.dump(list_of_employees, file)
72             for employee in list_of_employees:
73                 print(f"You have reviewed the employee, {employee['first_name']} {employee['last_name']}, on {employee['review_date']}, with a")
74     except Exception as error_message:
75         presentation_classes.IO.output_error_messages(f"There was an error saving the data to the {file_name} file!", error_message)
76
77 class IO: # Stores all functions that have to do with inputs/outputs
78     # Function to output error message whenever needed
79     @staticmethod
80     def output_error_messages(message: str, error: Exception = None):
81
82     # Function that prints the menu options
83     @staticmethod
84     def output_menu(menu: str): # Prints the menu of options...
85
86     # Function that stores the user's menu choice
87     @staticmethod
88     def input_menu_choice():
89
90     # Function that allows user to add employees to the data
91     @staticmethod
92     def input_employee_data(employee_data: list): # Adds user to database...
93
94     # Function that prints out the current data
95     @staticmethod
96     def output_employee_data(employee_data: list): # Presents all information to the user...
97
98     # Function that ends the program
99     @staticmethod
100    def quit_program(): # Ends the program...
```

Figure 9: Python Classes Example.

Objects and Inheritance

In programming, objects are classifications of specific data. These objects can have specific parameters that each instance must have. For example, in this assignment I use two class objects: Person and Employee. Any instance of a person must contain the parameters first_name and last_name. Since any employee is also a person, it inherits these required parameters as well as has two of its own: review_date and review_rating (Figure 10). I was not sure exactly how to use the review_date parameter, so I did some research online¹ and was able to find some useful code. I utilized these classes when the user chooses option 1, which allows them to input an employee into the data. Whatever information they input is then stored and used to create an instance of an Employee and by extension an instance of a Person.

```
10 class Person: # Creates the Person object
11
12     # Initializes the Person object
13 > def __init__(self, first_name: str, last_name: str):...
14
15
16
17 > def __str__(self):...
18
19
20 @property
21 > def first_name(self):...
22
23
24 @first_name.setter
25 > def first_name(self, value: str):...
26
27
28 @property
29 > def last_name(self):...
30
31
32 @last_name.setter
33 > def last_name(self, value: str):...
34
35
36
37
38
39
40
41
42
43
44
45 class Employee(Person): # Creates the Employee object which inherits from the Person class
46
47     # Initializes the Employee object
48 > def __init__(self, first_name: str = "", last_name: str = "", review_date: datetime.date = "1900-01-01", review_rating: int = 3):...
49
50
51
52
53 > def __str__(self):...
54
55
56 > def __repr__(self):...
57
58
59
60 @property
61 > def review_date(self):...
62
63
64 @review_date.setter
65 > def review_date(self, value):...
66
67
68 @property
69 > def review_rating(self):...
70
71
72 @review_rating.setter
73 > def review_rating(self, value):...
```

Figure 10: Python Objects/Inheritance Example.

Separate files

In another attempt to organize code neatly, different classes are now separated by file. I had four files to store my code: the data_classes.py file, the presentation_classes.py file, the processing_classes.py file, and the main.py file (Figure 11). To utilize functions and classes from other files I used the import feature.





 data_classes	3/10/2024 6:29 PM	Python File	4 KB
 main	3/8/2024 8:40 AM	Python File	2 KB
 presentation_classes	3/8/2024 8:43 AM	Python File	3 KB
 processing_classes	3/8/2024 8:43 AM	Python File	3 KB

Figure 11: Python Separate Files Example.

¹ <https://www.geeksforgeeks.org/python-validate-string-date-format/>

Unit Testing

Unit testing is a very important part of coding. Unit testing is used to ensure that all functions and classes are running properly and throw exceptions when necessary. The way unit testing works is when you compare what the code does to what you want it to do. If it does match, then the test will pass and if it does not match, then the test will fail. I used unit testing to ensure that each of my functions ran properly. For example, I wanted to ensure that the Person class was running correctly so I did some unit testing on it (Figure 12). I first checked that the initialization was working by creating an instance of a person, then asserting that the first and last name matched that of the instance I created. I proceeded to check that for the first and last name parameters as well.

```
# Testing the Person class
class TestPerson(unittest.TestCase):
    # Tests initialization of a Person
    def test_person_init(self):
        person = Person("John", "Doe")
        self.assertEqual(person.first_name, "John")
        self.assertEqual(person.last_name, "Doe")
    # Tests Person.first_name
    def test_person_first_name(self):
        person = Person("Jane", "Doe")
        self.assertEqual(person.first_name, "Jane")
    # Tests if exception is thrown with invalid first name
    def test_person_invalid_first_name(self):
        with self.assertRaises(ValueError):
            Person("123", "Doe")
    # Tests Person.last_name
    def test_person_last_name(self):
        person = Person("Jane", "Doe")
        self.assertEqual(person.last_name, "Doe")
    # Tests if exception is thrown with invalid first name
    def test_person_invalid_last_name(self):
        with self.assertRaises(ValueError):
            Person("John", "123")
```

Figure 12: Python Unit Testing Example.

Summary

In conclusion, this assignment utilized 12 topics: loops, conditional statements, lists, dictionaries, error handling, JSON files, functions, classes, objects, inheritance, file separation, and unit testing. I utilized a 'while' loop to continuously print the options from the menu to the user and I used a 'for' loop in order to relay to the user what task they completed in case three. I used conditional statements to complete a task based on the menu option the user chose. I used lists to store Employees given by the user. I used error handling to ensure that my program would not crash. I used the information the user inputted to create a JSON file with the information in it. I used functions, classes, and file separation to help organize and keep my program neat. I used objects and inheritance to create instances of Employees as opposed to storing their information in variables. Lastly, I used unit testing to ensure that my program was working as intended.