Aisha Shafique
06.09.2022
Foundations of Programming: Python
Assignment 09
https://github.com/Ai-Shaf/IntrotoProg-Python-Mod09

Script Modules & Inheritance

This week's assignment works with organizing code into script modules that can be imported into the main script. The three modules we had for importing were DataClasses, ProcessingClasses, and IOClasses.

In addition to learning about importing module scripts, we also covered inheritance of classes this week. All classes inherit from the Object class, which is the ultimate Base class in Python. Then, we can create classes that derive from other base classes that we create. In this class, we had created a class Person(), which was the parent, or super, class for our class Employee(). The Employee() class was derived from Person(), so it inherited all of the methods, including the constructor, from Person(). Now, I could choose to overwrite the derived methods if I wished, but by default, they are there for me to use as is. In this case, the listings we were given showed that the __str__() method was overwritten so that it would return ID, first name, and last name for an instantiated objected from the Employee Class. An instantiated object for the Person() class would have only returned first name and last name according to its string method.

Both the parent class Person() and child class Employee() make up our DataClasses module. The ProcessingClasses and IOClasses modules only have one class each for this assignment (though it has a suggestion that Database Processing could be added onto, it is not done so for this class).

After using the TestHarness file to test components of the modules, I use the Main script to import the three modules and then add in code to implement the pseudo-code provided by the instructor. Below are snapshots of the code for this assignment.

Writing the Script

The main script follows closely with the tasks that we have been setting to do in previous assignments. This time, however, we compartmentalize even more by importing the modules so that this file focuses solely on the main script.

```
# ------------------------------------------------------------------------ #
# Title: Assignment 09 Main
# Description: Working with Modules

# ChangeLog (Who,When,What):
# RRoot,1.1.2030,Created started script
# RRoot,1.1.2030,Added pseudo-code to start assignment 9
# AShafique,6.9.2022,Modified code to complete assignment 9
# ------------------------------------------------------------------------ #
# TODO: Import Modules
if __name__ == "__main__":
    import DataClasses as DC  # data classes
    import ProcessingClasses as P  # processing classes
    import IOClasses as IO  # IO classes
else:
    raise Exception("This file was not created to be imported")

# Main Body of Script  ---------------------------------------------------- #
# TODO: Add Data Code to the Main body. Done.
lstEmployees = []  # list for list of employee objects
strFile = "EmployeeData.txt"
strChoice = ""
strStatus = ""
lstObjects = []

# Load data from file into a list of employee objects when script starts
try:
    lstEmployees = P.FileProcessor.read_data_from_file(strFile)
    for item in lstEmployees:
        emp_obj = DC.Employee(item[0], item[1], item[2].strip())
        lstObjects.append(emp_obj)

except FileNotFoundError as e:
    print("File not found.")
except Exception as e:
    print(e, e.__doc__, type(e), sep='\n')

    # Show user a menu of options
while True:
    IO.EmployeeIO.print_menu_items()

# Get user's menu option choice
    strChoice = IO.EmployeeIO.input_menu_options()

    # Show user current data in the list of employee objects
    if strChoice == "1":
        IO.EmployeeIO.print_current_list_items(lstObjects)
        continue

    # Let user add data to the list of employee objects
    elif strChoice == "2":
        try:
            emp_data = IO.EmployeeIO.input_employee_data()
            print(emp_data, type(emp_data))
            lstObjects.append(emp_data)
        except Exception as e:
            print(e, e.__doc__, type(e))
```

```
        continue

    # let user save current data to file
    elif strChoice == "3":
        strStatus = P.FileProcessor.save_data_to_file(strFile, lstObjects)
        if strStatus == True:
            print("Data Save Success!")
        else:
            print("Data NOT Saved.")
        continue

    # Let user exit program
    elif strChoice == "4":
        print("Exiting program.")
        break

    else:
        print("Please only choose 1, 2, 3, or 4!")

# Main Body of Script  ---------------------------------------------------- #
```

Listing 1: Main Script Code

I start with the if __name__ = __main__: condition before I import the modules; then I list the main global variables I will be using, and lastly, I proceeded to implement the pseudo-code. My first condition ensures that this file is used as a main script and not as a module script.

I used try/except blocks when loading the code into the table so that I could handle errors resulting from malfunction of data being loaded –an instance of interacting with the outside. Similarly, I added try/except blocks when the user inputs new employee information, as I'm interacting with the outside and want to catch possible exceptions that may occur due to user error.

For menu choice, I don't use try/except, but stick with the if/elif/else statements. My else is a catch all so that if the user enters anything but the 4 choices, s/he is reminded to stick to the menu options.

After loading data from the read_data_to_file() function from my FileProcessor class, I turn the list of lists into list of objects and simultaneously strip away line breaks. They were not stripped in the original function, which was leading to '\n' showing up when the data was loaded and displayed.

I also had to modify from the IOClasses the function to print current tasks. This function failed to perform when being tested in the TestHarness mini script. I simplified it so that it would function by simply using row.employee_id (or first_name or last_name attributes). Since I'm importing from a list of objects, each row is an object and I can call the employee.id method without restating the module or class.

After these changes, the code ran fairly smoothly.

```
@staticmethod
def print_current_list_items(list_of_rows: list):
    """ Print the current items in the list of Employee rows

    :param list_of_rows: (list) of rows you want to display
```

```
    """

    print("******* The current items employees are: *******")
    for row in list_of_rows:
        print(str(row.employee_id)
              + ","
              + row.first_name
              + ","
              + row.last_name)
    print("*****************************************")
```

Listing 2: Updated print_current_list_items method

Running the Code

I run the code in both PyCharm and the Windows Command Console. Figures 1 and 2 below show the code running.

The code running in PyCharm in Figure 1 simply shows what the list of Employees looks like when I display it.

The code running the Command Console shows user adding employee ID, first and last name to create another data entry.

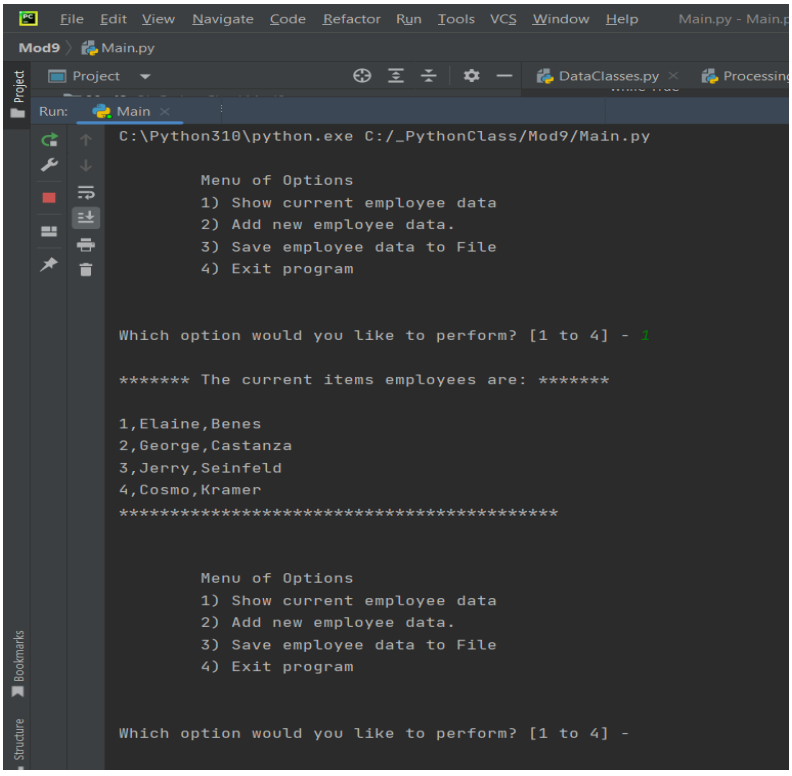

Figure 1: Assignment 09 Main.py running in PyCharm

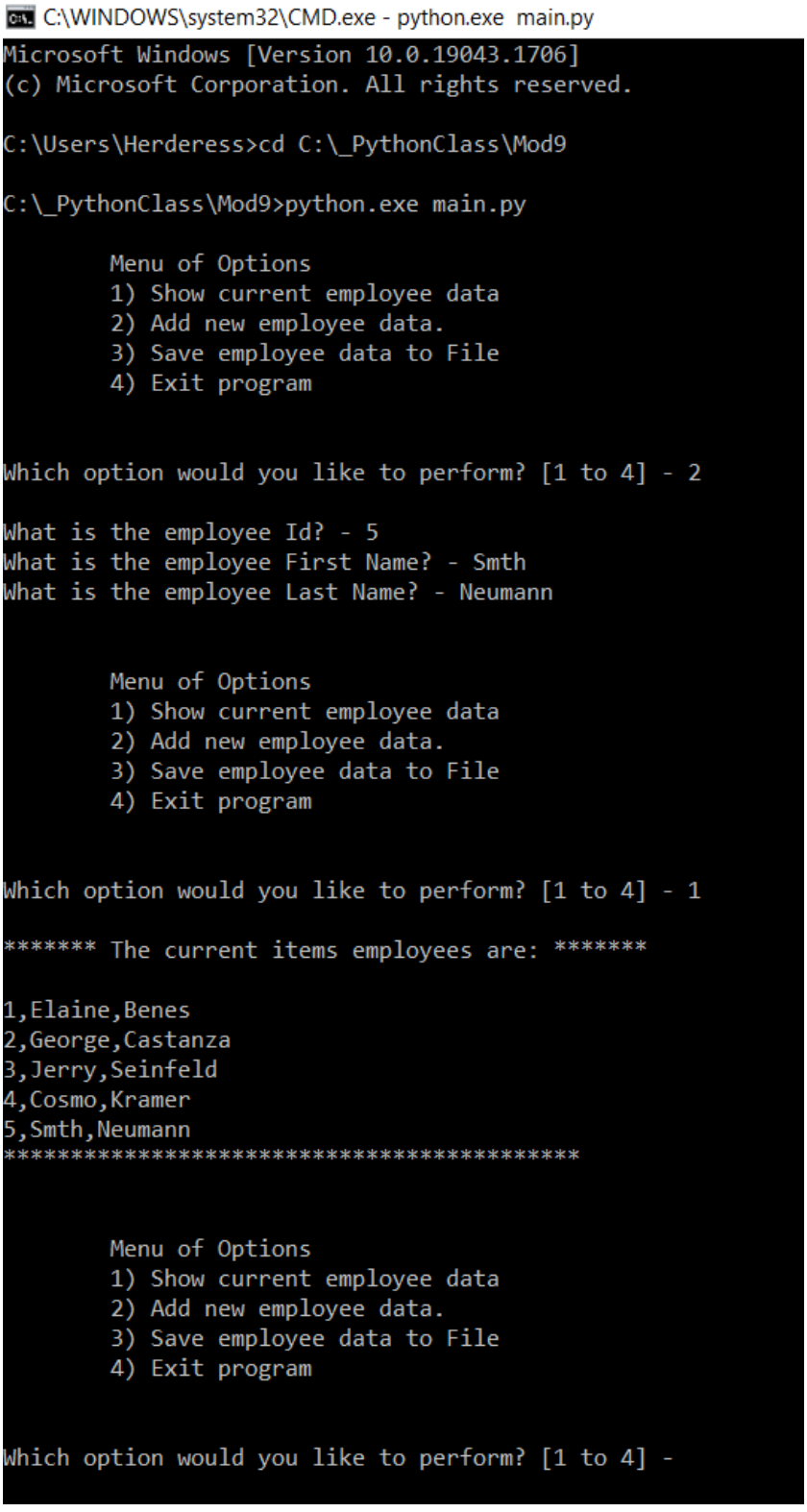

Figure 2: Main Script for Assignment 09 running in Windows OS Shell

Summary

In this assignment, we learned how class inheritance works. We then grouped our classes into script modules that we saved in separate files. We then used a testharness file to import the modules to test them one at a time, before we created a main script file in which to enter code for our application. We then ran our code in PyCharm and the Windows Command Prompt to ensure that our file was running correctly.