Alicia Davis
27 May 2024
IT FDN 110 B
Assignment 07

# Data Classes & Inheritance

## Introduction

This assignment builds on last week's assignment of creating a menu program. Instead of the data being saved as a list of dictionary rows, it's being saved and presented as a list of object rows. The code along with this knowledge document is also hosted on my GitHub repository here.

## Creating the Script

Similar to the last assignments, I start my script off with a header, importing the exit and JSON functions and defining the constants and variables. The constants and global variables stayed the same as last week as local variables will be defined as instances within two new classes (Figure 1.1).

```
import json
from sys import exit

# Constants and global variables
MENU: str = """
    ---- Course Registration Program ----
    Select from the following menu:
    1. Register a student for the course
    2. Show current data
    3. Save data to file
    4. Exit the program
    ----------------------------------------
"""
FILE_NAME: str = "enrollments.json"

menu_choice: str = ""
students: list = []
```

**Figure 1.1: Updated import functions, constants and global variables.**

Next I defined the data layer / file processing block of my code. This area contains three classes: Person, Student and FileProcessor. Person and Student are data classes that will create instances of the data with Student inheriting attributes from Person. FileProcessor contains functions that will be leveraged to process my JSON file (Figure 1.2).

```
# Data layer / File processing block
class Person:···


class Student(Person):···


class FileProcessor:···
```

**Figure 1.2: Person, Student & FileProcessor classes.**

The first class is Person, which I created to house the student's first and last name, and uses Python's constructor __init__ to create an object of the class. Each instance of the object is identified using "self" and there are two functions, both with getters and setters for first and last name. The attributes within this class are also defined as private indicated with the double underscore. I've also redefined the __str__ method to include the class instance's attributes to show a cleaner stringer representation. Additionally, I added error handling on the instance itself to ensure that only alphanumeric values are valid. This will be utilized later in the IO class when I ask for user input (Figure 1.3).

```
class Person:
    """Defines parent class to house student's first and last names"""

    def __init__(self, student_first_name: str = "", student_last_name: str = ""):
        self.student_first_name = student_first_name
        self.student_last_name = student_last_name

    @property
    def student_first_name(self):
        return self.__student_first_name.title()

    @student_first_name.setter
    def student_first_name(self, value: str):
        if value.isalpha():
            self.__student_first_name = value
        else:
            raise ValueError("First name can't contain non-alphanumeric values.")

    @property
    def student_last_name(self):
        return self.__student_last_name.title()

    @student_last_name.setter
    def student_last_name(self, value: str):
        if value.isalpha():
            self.__student_last_name = value
        else:
            raise ValueError("Last name can't contain non-alphanumeric values.")

    def __str__(self):
        return f"{self.student_first_name},{self.student_last_name}"
```

**Figure 1.3: Person data class with constructor and built in error-handling logic.**

The next class is Student, which is inheriting attributes from the Person class in addition to adding a new attribute for course name. This is structured similarly to the Person class, however since it's inheriting the student's first and last name, I included the super function to map the values from Person to Student. The error handling also raises an error if the course value is empty (Figure 1.4).

```
class Student(Person):
    """Defines child class of Person and adds course name"""

    def __init__(self, student_first_name: str, student_last_name: str, course_name: str = ""):
        super().__init__(student_first_name=student_first_name, student_last_name=student_last_name)
        self.course_name = course_name

    @property
    def course_name(self):
        return self.__course_name

    @course_name.setter
    def course_name(self, value: str):
        if len(value) != 0:
            self.__course_name = value
        else:
            raise ValueError("Course name can't be empty.")

    def __str__(self):
        return f"{self.student_first_name},{self.student_last_name},{self.course_name}"
```

**Figure 1.4: Student subclass with constructor and built in error-handling logic.**

Similar to last week, the FileProcessor class has two functions. The first is for reading data from a file, while the second is for writing data to a file. The difference from last week's assignment is that both methods are leveraging the Student class's instances to read/write the data to/from a list of object rows (Figure 1.5).

```python
class FileProcessor:
    """Class to handle data storage and retrieval"""

    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """Read file and load to json and return list"""

        try:
            with open(file_name) as file:
                list_of_dict_data = json.load(file)
                for student in list_of_dict_data:
                    student_object: Student = Student(student_first_name=student["FirstName"],
                                                      student_last_name=student["LastName"],
                                                      course_name=student["CourseName"])
                    student_data.append(student_object)
        except FileNotFoundError as error_message:
            IO.output_error_messages("\nFile not found.", error_message)
        except Exception as error_message:
            IO.output_error_messages("\nUnknown Error. Please contact support. ", error_message)
        return student_data

    @staticmethod
    def write_data_to_file(file_name: str, student_data: list):
        """Write data to json file"""

        try:
            list_of_dict_data: list = []
            for student in student_data:
                student_json: dict = {"FirstName": student.student_first_name,
                                      "LastName": student.student_last_name,
                                      "CourseName": student.course_name
                                      }
                list_of_dict_data.append(student_json)
            with open(file_name, "w") as file:
                json.dump(list_of_dict_data, file)
        except FileNotFoundError as error_message:
            IO.output_error_messages("\nFile not found.", error_message)
        except Exception as error_message:
            IO.output_error_messages("\nUnknown Error. Please contact support.", error_message)
```

**Figure 1.5: Read and write functions under the FileProcessor class with built in error-handling logic.**

The next class is the presentation layer or IO (input/output) block. This contains all of the code that will get input from and give output to a user. Each function doesn't change the class and is therefore marked as "staticmethod" (Figure 1.6).

```python
# Presentation Layer / IO block
class IO:
    """Class to handle user input and output"""

    @staticmethod
    def output_error_messages(message: str, error: Exception = None):...

    @staticmethod
    def output_menu(menu: str):...

    @staticmethod
    def input_menu_choice():...

    @staticmethod
    def input_student_data(student_data: list):...

    @staticmethod
    def output_student_courses(student_data: list):...
```

**Figure 1.6: IO class containing five functions.**

The first three IO functions didn't change from last week's assignment. The input_student_data function changed from last week in two key ways: 1) the input data is being stored as objects of the Student class instead of dictionary rows and 2) the error handling has moved to the Person & Student classes instead of being handled as part of the user input (Figure 1.7).

```
@staticmethod
def input_student_data(student_data: list):
    """Process user's input and append to dictionary"""

    try:
        student_first_name = input("Enter the student's first name: ")
        student_last_name = input("Enter the student's last name: ")
        course_name = input("Enter the course name: ")

        new_student = Student(student_first_name, student_last_name, course_name)

        student_data.append(new_student)
        print()
        print(f"You have registered {student_first_name} {student_last_name} for {course_name}.")
    except ValueError as error_message:
        IO.output_error_messages("\nInvalid Entry.  See details below.", error_message)
    except Exception as error_message:
        IO.output_error_messages("\nUnknown Error. Please contact support.", error_message)
    return student_data
```

**Figure 1.7: Input student data function with error handling.**

The output_student_courses function changed from last week and instead of printing the data with an f-string using dictionary keys, it prints based on the Student classes object attributes (Figure 1.8).

```
@staticmethod
def output_student_courses(student_data: list):
    """Output current data to user"""

    print("-"*50)
    print("The current data is: ")
    for student in student_data:
        print(student.student_first_name, student.student_last_name, student.course_name)
    print("-"*50, end="\n\n")
```

**Figure 1.8: Output student data function.**

The last block of code is the processing layer or the execution block. This is indicated by both my comment and the "__main__" name which is used to define the top-level of the code. This code didn't change from last week's assignment (Figure 1.9).

```
# Processing Layer / Execution block
if __name__ == "__main__":
    students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)

    while True:
        IO.output_menu(MENU)
        menu_choice = IO.input_menu_choice()
        match menu_choice:
            case "1":
                IO.input_student_data(student_data=students)

            case "2":
                IO.output_student_courses(student_data=students)

            case "3":
                FileProcessor.write_data_to_file(file_name=FILE_NAME, student_data=students)
                print("INFO: Registrations have been saved.")

            case "4":
                print("Program Ended.")
                exit()
```

**Figure 1.9: Execution block of code pulling in all functions across classes and defining the arguments for each function's parameters.**

# Testing the Script

I tested my code in both VSCode's terminal and IDLE. My testing followed the same steps as last week. I checked that my error messages for the file not existing (Figure 2.1) and invalid options entered as part of menu option 1 worked as expected (Figure 2.2). Because the error handling is done on the Person/Student classes rather than IO class, the error message is not displayed until after the input function completes even if the error occurred while entering the first name.

```
PS C:\Users\adavi\OneDrive\Documents\UW\FDN110 - Foundations of Python\Module07\Assignment> python assignment07.py

File not found.

--- Error Details ---
[Errno 2] No such file or directory: 'enrollments.json'
File not found.
<class 'FileNotFoundError'>
```

**Figure 2.1: File not found error message when launching program.**

```
Choose a menu option (1-4): 1
Enter the student's first name: 1
Enter the student's last name: Davis
Enter the course name: FDN 110

Invalid Entry.  See details below.

--- Error Details ---
First name can't contain non-alphanumeric values.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

```
Choose a menu option (1-4): 1
Enter the student's first name: Alicia
Enter the student's last name: 2
Enter the course name: FDN 110

Invalid Entry.  See details below.

--- Error Details ---
Last name can't contain non-alphanumeric values.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

```
Choose a menu option (1-4): 1
Enter the student's first name: Alicia
Enter the student's last name: Davis
Enter the course name:

Invalid Entry.  See details below.

--- Error Details ---
Course name can't be empty.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

**Figure 2.2: Error messages for blank and non-alphanumeric inputs for menu option 1.**

Once valid data is entered, menu option 2 displays the current data (Figure 1.3).

```
Choose a menu option (1-4): 2
-------------------------------------------------
The current data is:
Bob Smith FDN 110
Sue Jones FDN 110
Alicia Davis FDN 110
Michael Jordan FDN 110
Charles Barkley FDN 110
-------------------------------------------------
```

**Figure 2.3: Current data in the "student" list.**

Menu option 3, creates the enrollments.json file and saves the new user input to the file (Figure 2.4).

```
Assignment > {} enrollments.json > ...
  1  {"FirstName": "Alicia", "LastName": "Davis", "CourseName": "FDN 110"}, {"FirstName": "Michael", "LastName": "Jordan", "CourseName": "FDN 110"}, {"FirstName": "Charles", "La
```

**Figure 2.4: JSON file with data from user input.**

Any input outside 1-4 still results in an invalid option message (Figure 2.5), and option 4 still ends the program (Figure 2.6).

```
Choose a menu option (1-4): 5


--- Error Details ---
Invalid option.  Please choose between 1-4.
Common base class for all non-exit exceptions.
<class 'Exception'>
```

**Figure 2.5: Invalid option**

```
Choose a menu option (1-4): 4
Program Ended.
```

**Figure 2.6: Program Ended**

Similar to in VSCode's terminal, my testing followed the same steps in IDLE, except since the file exists from my testing in terminal, I don't receive the error message that the file can't be found. I do receive the correct error message for invalid inputs in menu option 1 though (Figure 2.7).

```
Choose a menu option (1-4): 1
Enter the student's first name: P3nny
Enter the student's last name: Hardaway
Enter the course name: FDN 110

Invalid Entry.  See details below.

--- Error Details ---
First name can't contain non-alphanumeric values.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

```
Choose a menu option (1-4): 1
Enter the student's first name: Penny
Enter the student's last name: H4rdaway
Enter the course name: FDN 110

Invalid Entry.  See details below.

--- Error Details ---
Last name can't contain non-alphanumeric values.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

```
Choose a menu option (1-4): 1
Enter the student's first name: Penny
Enter the student's last name: Hardaway
Enter the course name:

Invalid Entry.  See details below.

--- Error Details ---
Course name can't be empty.
Inappropriate argument value (of correct type).
<class 'ValueError'>
```

**Figure 2.7: Error messages for blank and non-alphanumeric inputs for menu option 1.**

Once valid data is entered, I can see these users were added to my master list when selecting option 2 (Figure 2.8).

```
Choose a menu option (1-4): 2
--------------------------------------------------
The current data is:
Bob Smith FDN 110
Sue Jones FDN 110
Alicia Davis FDN 110
Michael Jordan FDN 110
Charles Barkley FDN 110
Penny Hardaway FDN 110
Scottie Pippin FDN 110
Karl Malone FDN 110
--------------------------------------------------
```

**Figure 2.8: New users added to master list.**

Option 3 saves the data as comma-delimited to the existing csv file (Figure 2.9 & Figure 2.10).

```
Choose a menu option (1-4): 3
INFO: New registrations have been saved to file.
```

**Figure 2.9: Message to user that new input has been saved.**

Assignment > {} enrollments.json > ...


```
1  "FirstName": "Penny", "LastName": "Hardaway", "CourseName": "FDN 110"}, {"FirstName": "Scottie", "LastName": "Pippin", "CourseName": "FDN 110"}, {"FirstName": "Karl", "La
```


**Figure 2.10: Updated JSON File**

Any input outside 1-4 still results in an invalid option message (Figure 2.11), and option 4 still ends the program (Figure 2.12).

```
Choose a menu option (1-4): 5



--- Error Details ---
Invalid option.  Please choose between 1-4.
Common base class for all non-exit exceptions.
<class 'Exception'>
```

**Figure 2.11: Invalid option**

```
Choose a menu option (1-4): 4
Program Ended
>>>
```

**Figure 2.12: Program Ended**

# Summary

This week I started with my assignment 06 code as a base instead of starting from scratch. I think this may have made it more difficult because I ran into quite a few errors that had me going to Google to figure out. The main one was around the two new data classes: Person and Student. The major error was a “maximum recursion depth exceeded” which I found out was because I had not created private attributes in Person and Student, but instead created a function inside of a function which was calling itself recursively. Once I fixed the attribute so it was considered a variable instead of a function, that solved my error. I also ran into an error with my input_student_data function as I was adding new instances of the object to Student. I solved this by leveraging what Luis showed in class and created a separate variable for a new student and equating it to the Student variables, then appending that onto my student_data.