August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link

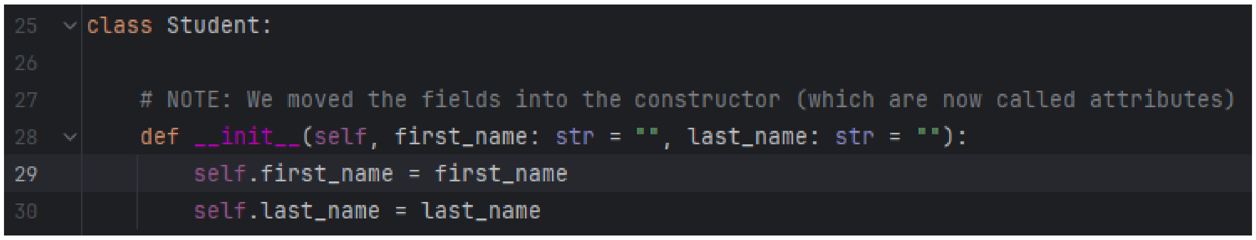
Inherited Classes, Constructors, and Properties

Introduction

Module 07 continues to build on the program and lessons from the previous few modules. We rebuilt Assignment06's program to use inherited classes for user inputs and utilize constructors and properties. The overall function of the program remains the same to the user. This assignment highlights what we learned in the module regarding inherited classes, constructors, and properties. Inherited classes allow us to extend upon a class and give it multiple implementations where classes can inherit or share attributes. We learned how to use constructors to initialize an object of a class and how to use properties to work with instances, additional logic, and better data validation. All of these lessons are showcased in Assignment07.

Classes and Constructors

In this section of the module, we adapted classes to turn input results into instances for variables that get added to a list. These variables become class **attributes** or variables of a class that get shared between all instances of that class. Using our existing demo examples, the "Student" input variables can become their own class object where they're reusable, instead of getting appended to a dictionary. Then, we use constructors to create data objects with an initial set of values. A **constructor** is a special method in a class used to create and initialize an object of a class in the order we want for presenting the data. Without a constructor, the class has no initial value specifying that a "Student" entry, for example, should include a first name and a last name. The constructor can be used over and over again to continue to add instances to a given roster or list and won't be created unless all required attributes are input. Constructors automatically get invoked when an object of a class is created. A constructor must begin with the "def" keyword, like all functions, and must be followed by the "_ _init_ _" prefix. It takes an argument called "self", which assigns values to the variables and refers to the individual instance. Then, parameters get called in parentheses like any other method. **(See Figure A)**



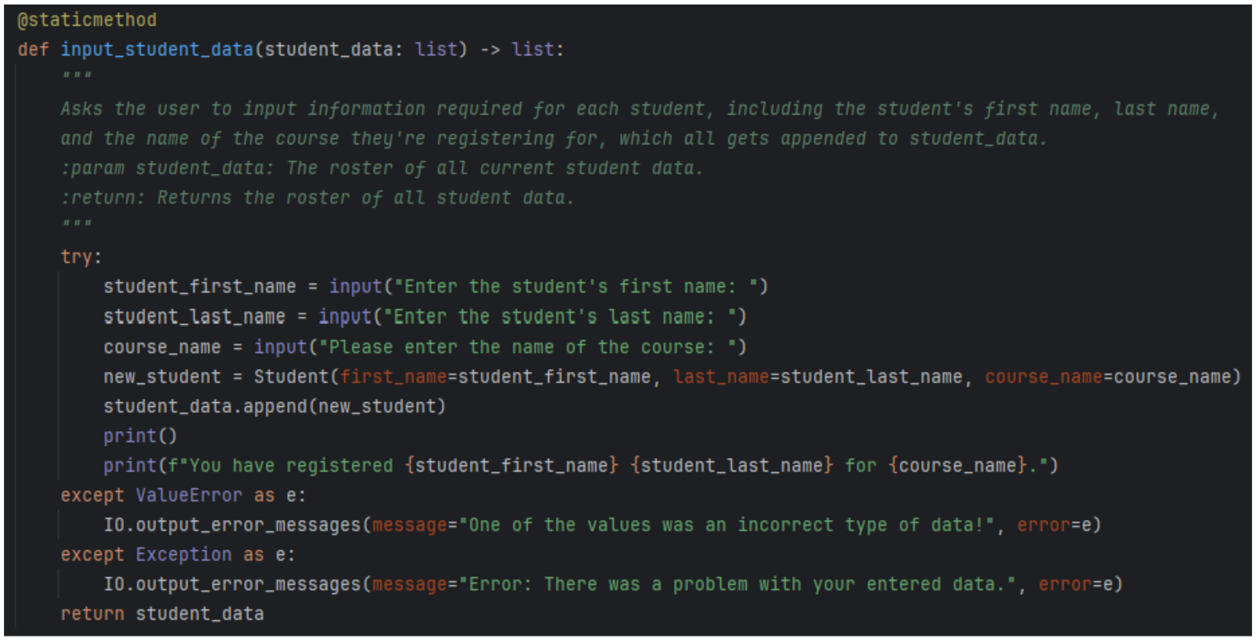
```
class Student:

    # NOTE: We moved the fields into the constructor (which are now called attributes)
    def __init__(self, first_name: str = "", last_name: str = ""):
        self.first_name = first_name
        self.last_name = last_name
```

Figure A: Example code of a constructor under the class "Student" that includes attributes for a first name and a last name.

August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link

Getting the user's input for this data can now have paired down code when called in an input function later in the program. Each new instance of data gets appended to the full roster of student data. **(See Figure B)**

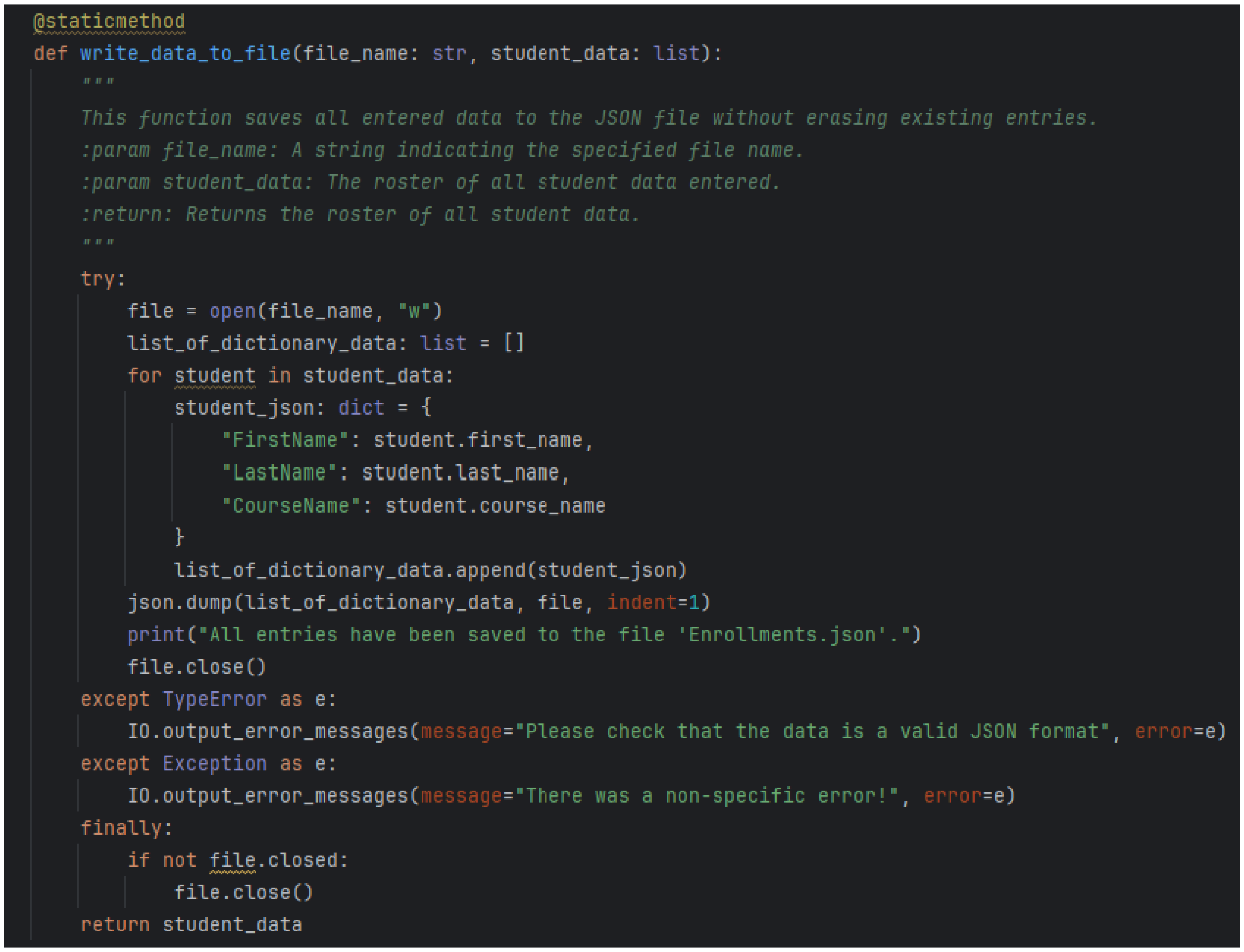


```
@staticmethod
def input_student_data(student_data: list) -> list:
    """
    Asks the user to input information required for each student, including the student's first name, last name,
    and the name of the course they're registering for, which all gets appended to student_data.
    :param student_data: The roster of all current student data.
    :return: Returns the roster of all student data.
    """
    try:
        student_first_name = input("Enter the student's first name: ")
        student_last_name = input("Enter the student's last name: ")
        course_name = input("Please enter the name of the course: ")
        new_student = Student(first_name=student_first_name, last_name=student_last_name, course_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 e:
        IO.output_error_messages(message="One of the values was an incorrect type of data!", error=e)
    except Exception as e:
        IO.output_error_messages(message="Error: There was a problem with your entered data.", error=e)
    return student_data
```

Figure B: Code from Assignment07 that gathers the user's input for instanced data attributes for the subclass "Student".

If this data needs to be written to or read from a JSON file, then the formatting needs to change from programs we've worked on in previous modules because JSON files are set up in correct dictionary formatting by default, but data classes do not use the dictionary format. We need to modify the load function and create a new list object to accept a different variable that we've appended our class instances to instead. **(See Figures C and D)**

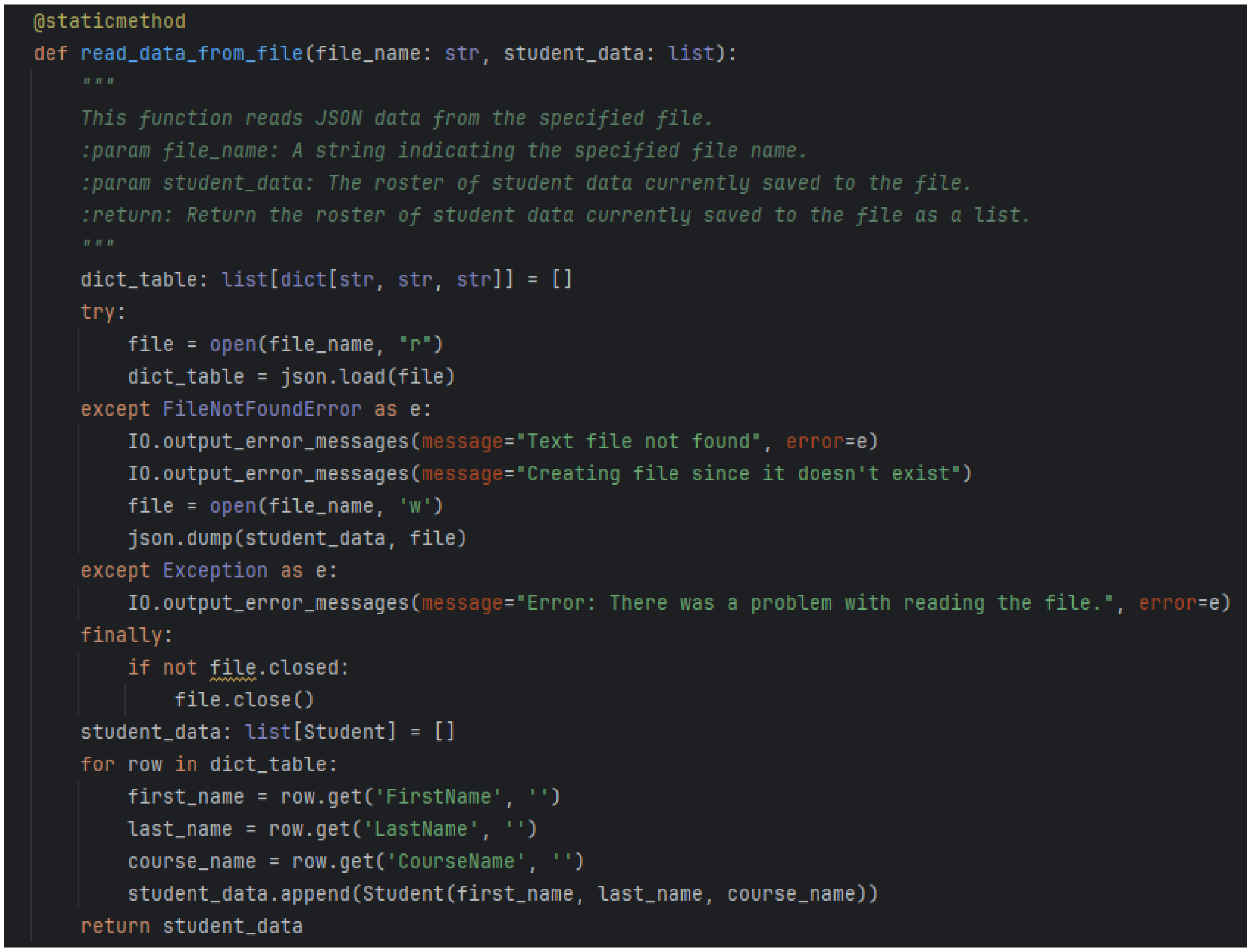
August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link



```
@staticmethod
def write_data_to_file(file_name: str, student_data: list):
    """
    This function saves all entered data to the JSON file without erasing existing entries.
    :param file_name: A string indicating the specified file name.
    :param student_data: The roster of all student data entered.
    :return: Returns the roster of all student data.
    """
    try:
        file = open(file_name, "w")
        list_of_dictionary_data: list = []
        for student in student_data:
            student_json: dict = {
                "FirstName": student.first_name,
                "LastName": student.last_name,
                "CourseName": student.course_name
            }
            list_of_dictionary_data.append(student_json)
        json.dump(list_of_dictionary_data, file, indent=1)
        print("All entries have been saved to the file 'Enrollments.json'.")
        file.close()
    except TypeError as e:
        IO.output_error_messages(message="Please check that the data is a valid JSON format", error=e)
    except Exception as e:
        IO.output_error_messages(message="There was a non-specific error!", error=e)
    finally:
        if not file.closed:
            file.close()
    return student_data
```

Figure C: Code from Assignment07 showing how to write data from a data class to a JSON file.

August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link



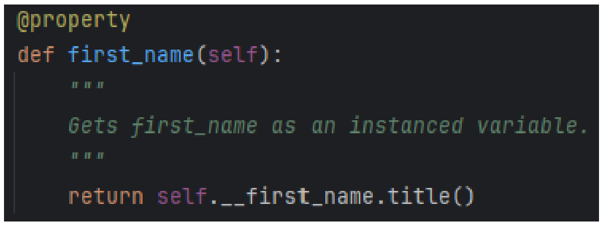
```python
@staticmethod
def read_data_from_file(file_name: str, student_data: list):
    """
    This function reads JSON data from the specified file.
    :param file_name: A string indicating the specified file name.
    :param student_data: The roster of student data currently saved to the file.
    :return: Return the roster of student data currently saved to the file as a list.
    """
    dict_table: list[dict[str, str, str]] = []
    try:
        file = open(file_name, "r")
        dict_table = json.load(file)
    except FileNotFoundError as e:
        IO.output_error_messages(message="Text file not found", error=e)
        IO.output_error_messages(message="Creating file since it doesn't exist")
        file = open(file_name, 'w')
        json.dump(student_data, file)
    except Exception as e:
        IO.output_error_messages(message="Error: There was a problem with reading the file.", error=e)
    finally:
        if not file.closed:
            file.close()
    student_data: list[Student] = []
    for row in dict_table:
        first_name = row.get('FirstName', '')
        last_name = row.get('LastName', '')
        course_name = row.get('CourseName', '')
        student_data.append(Student(first_name, last_name, course_name))
    return student_data
```

Figure D: Code from Assignment07 showing how to read data from a JSON file into a data class's roster.

Properties

A **property** is a way to get and set instances that allow us to include extra data validation or extra logic when accessing instances, which gives every instance of that variable appropriate error messaging that only needs to be typed out once. Properties are methods that begin with "@property" and should be named something that mirrors what the variable should be called when it's used. This is known as a "getter". Any new parameters added will apply to every instance that that property gets called, such as using ".capitalize()" or ".title()" to ensure every instance of that entry is returned in the proper case. **(See Figure E)**

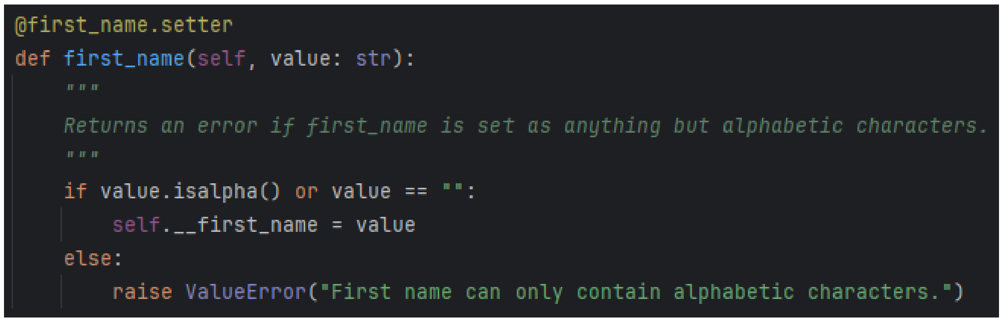
August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link



```
@property
def first_name(self):
    """
    Gets first_name as an instanced variable.
    """
    return self.__first_name.title()
```

Figure E: Example code of a property "first_name".

You can also use properties to raise ValueErrors in "setter" properties so that any time a value is set for that instance, it ensures the input values are of the appropriate type. Setters do not include any underscores at the start of the attribute's name so that it's turned from a variable into a setter. The setter property will get called any time you interact with that value, so whether the values are being input for the first time or read back from a JSON, error validation will run. **(See Figure F)**



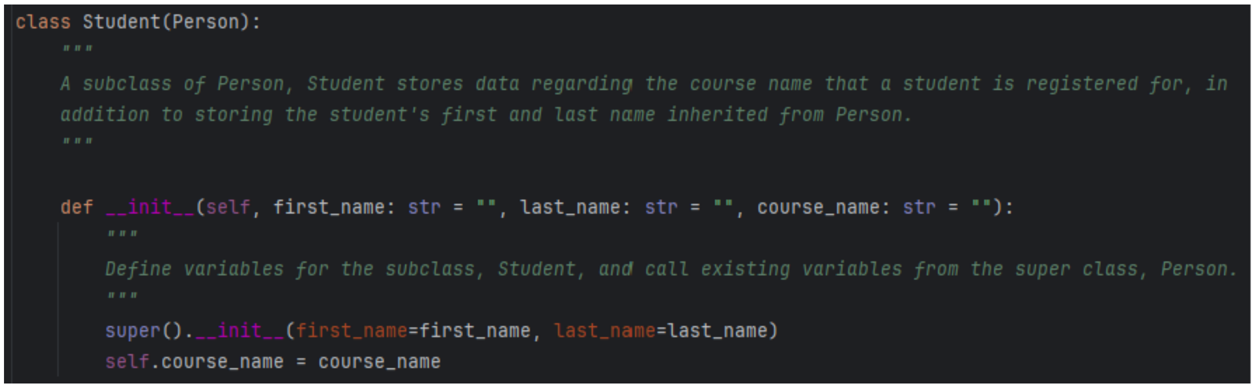
```
@first_name.setter
def first_name(self, value: str):
    """
    Returns an error if first_name is set as anything but alphabetic characters.
    """
    if value.isalpha() or value == "":
        self.__first_name = value
    else:
        raise ValueError("First name can only contain alphabetic characters.")
```

Figure F: Example code of creating a setter property "first_name.setter" that raises a ValueError to make sure that data entered for "first_name" only contains alphabetic characters.

Inheritance

Class Inheritance is the idea that a class can be extended upon to have multiple implementations. It allows you to create a base class and then extend that class to add additional information in subclasses. For the demos, "Student" became a subclass of "Person" because if we were to increase the registration roster to include "Teachers", there would be common information between a student and a teacher that we could avoid having to implement twice, such as the first name and last name attributes. "Person" becomes the super, or parent, class written like "class Person:" and the inherited class becomes "class Student(Person):". In inherited classes, you can call back to variables in the super class under the initialized method because if you create a student, you are also constructing a person. These variables get initialized from the super in the subclass's constructor. **(See Figure G)**

August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link

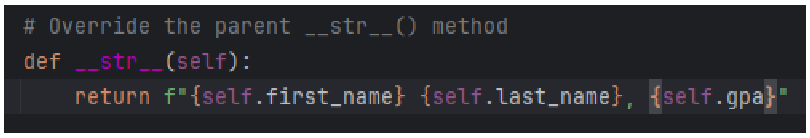


```python
class Student(Person):
    """
    A subclass of Person, Student stores data regarding the course name that a student is registered for, in
    addition to storing the student's first and last name inherited from Person.
    """

    def __init__(self, first_name: str = "", last_name: str = "", course_name: str = ""):
        """
        Define variables for the subclass, Student, and call existing variables from the super class, Person.
        """
        super().__init__(first_name=first_name, last_name=last_name)
        self.course_name = course_name
```

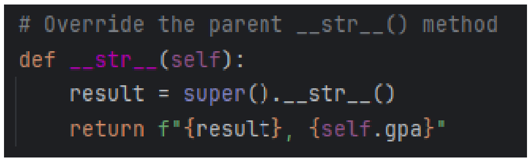
Figure G: Example of setting up an inherited subclass called "Student".

Methods from the super class can be overridden in the subclass by implementing an identical method in the inherited class. There are two ways to format this. The first is to duplicate the method completely and add whatever additional data you want to include for the inherited class. **(See Figure H.1)** The second way is to call the super class's method by getting the "result" of that method and then adding the additional data. **(See Figure H.2)** Both formats here would have the same outcome.



```python
# Override the parent __str__() method
def __str__(self):
    return f"{self.first_name} {self.last_name}, {self.gpa}"
```

Figure H.1: Example code of overriding a super class's method in a subclass by making an identical method in the subclass with the desired additional information to include "self.gpa".



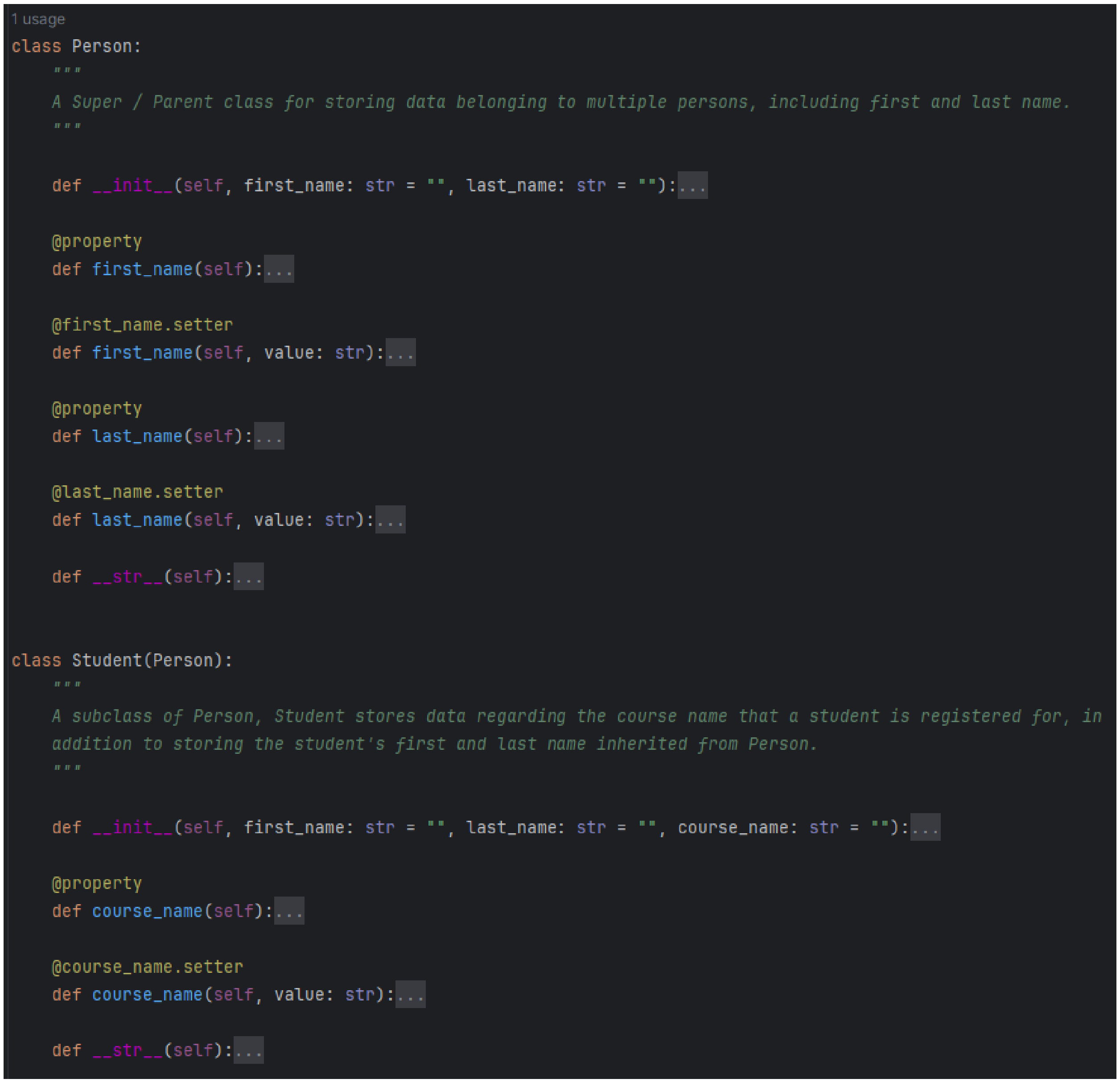
```python
# Override the parent __str__() method
def __str__(self):
    result = super().__str__()
    return f"{result}, {self.gpa}"
```

Figure H.2: Example code of overriding a super class's method in a subclass by calling the result of the super class's string method and returning that result with additional information from the subclass.

Creating the Script for Assignment 07

The goal of this assignment was to change the script from Assignment06 to use inherited classes for user inputs and utilize constructors and properties. The overall function of the program should remain the same to the user. I began the assignment by setting up the file, header, data constants, and variables according to the instructions and the starter file. Then, I

August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link
set up my inherited data classes so that there is a parent or super class called "Person" and a subclass called "Student". The class "Person" includes properties for a first name and a last name, as well as setters for error handling the inputs of these attributes. The subclass, "Student", inherits these attributes and constructs an additional attribute specific to students, which is the name of the course that that student is registering for. "Person" has a string method for returning the first and last name values. "Student(Person)" has a method to override this parent's method to include both of the variables from the super class and the additional course name variable from the student class. **(See Figure I)**



```
1 usage
class Person:
    """
    A Super / Parent class for storing data belonging to multiple persons, including first and last name.
    """

    def __init__(self, first_name: str = "", last_name: str = ""):...

    @property
    def first_name(self):...

    @first_name.setter
    def first_name(self, value: str):...

    @property
    def last_name(self):...

    @last_name.setter
    def last_name(self, value: str):...

    def __str__(self):...


class Student(Person):
    """
    A subclass of Person, Student stores data regarding the course name that a student is registered for, in
    addition to storing the student's first and last name inherited from Person.
    """

    def __init__(self, first_name: str = "", last_name: str = "", course_name: str = ""):...

    @property
    def course_name(self):...

    @course_name.setter
    def course_name(self, value: str):...

    def __str__(self):...
```

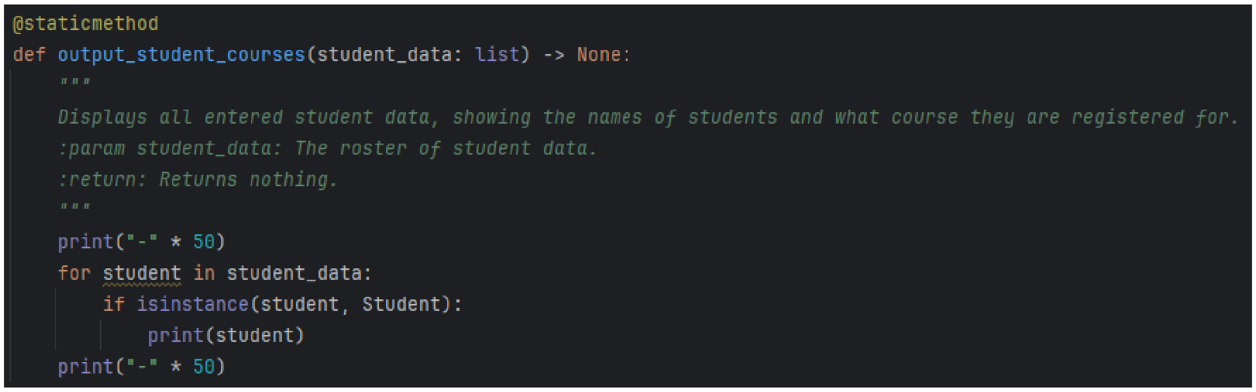
August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link

Figure I: Collapsed code that shows the different constructors and properties used in the "Person" and "Student" data classes in Assignment07.

The rest of the program is very similar to Assignment06. There is a processing class called "FileProcessor" and a presentation class called "IO" for input and output. "FileProcessor" has static methods for reading data from the JSON file and writing data to the file. While these methods and the class are ultimately set up the same as in Assignment06, the formatting of both of these methods needed to change in order to work with data class variables instead of dictionaries. **(See Figures C and D)**

In the presentation class "IO", we reuse all of the same methods from Assignment06 again. There is a static method for outputting error messaging and technical information for exceptions called "output_error_messages", which I left the same. The static method for inputting the user's menu choice for menu options 1, 2, 3, or 4, was also left the same as in Assignment06. I combined the methods for outputting the menu and inputting the user's choice in the method "input_menu_choice" in order to have more succinct code.

The next two static methods handle how the program should input the user's data and how the program should display all of the data back to the user. The formats for both of these methods needed to change in order to work with data classes instead of dictionaries. In order to output the roster of all current student data in the static method "output_student_courses", I used a for statement to look for a single instance of "student" saved in the list "student_data" and asked the program to print each instance of student from the class Student. **(See Figure J)** In order to input the user's data into instances of Student in "input_student_data", I first gather the user's input and then add that input to the data class's variables and append it to my overall list of "student_data" **(See Figure B)**. The error validation from the setter properties will handle any incorrect data entries for the first and last name, but the course name attribute is open to take entries of any value type, such as numbers, letters, or symbols.



```
@staticmethod
def output_student_courses(student_data: list) -> None:
    """
    Displays all entered student data, showing the names of students and what course they are registered for.
    :param student_data: The roster of student data.
    :return: Returns nothing.
    """
    print("-" * 50)
    for student in student_data:
        if isinstance(student, Student):
            print(student)
    print("-" * 50)
```

Figure J: Code from Assignment07 that outputs the roster of all current student data in the static method "output_student_courses".

August Belhumeur
11/29/2023
IT FDN 110
Assignment 07 - Document Knowledge
Github Link

The remaining program is largely the same from the starter file, with some additional edits and organization to make sure that all functions are called with the correct names and parameters. The biggest challenge for me in this assignment was tracking similarly named variables with the new data classes (i.e. “student”, “students”, “Student”, “student_data”, etc.) and making sure that I understood how each of these entities was functioning and interacting in the code and how to format it correctly for the data class. While editing the processing class code, I found that the data wasn’t reading from the file, but also not calling an error. I utilized ChatGPT to provide me with debug error messaging that PyCharm was not providing. This helped me to find what was missing from my code so that I could edit it, include better error validation, and continue testing in PyCharm and the Command Line.

Summary

In this module, we learned about inherited classes, constructors, and properties. Inherited classes allowed us to create a parent and subclass for the input variables in Assignment07, where the subclass can inherit attributes from the parent. We used constructors to initialize objects of a class and turned the class’s attributes into properties in order to accept inputs as instances. Lastly, we learned how to re-format the processing and presenting classes in Assignment07 to use and manipulate the information now stored in instances of the Student data class.