Olivier Richer
June 7, 2025, IT FDN 110 A Sp 25: Foundations of Programming: Python
GitHub hyperlink: olik2000 · GitHub
Assignment 07- Classes and Objects.

Classes and Objects.

Introduction:

Assignment #7 scope of work is to create a Python program using constants, variables, and print statements to display a message about a student's registration for a Python course. This program is very similar to Assignment06, but **It adds set of data classes.**

The script.

Per Becky 'request, I will not download the whole program. However, we will show the script pertaining to class person, and class student in which the extensive comments are embedded with the code. We will also show the code pertaining to the read_data_from_file () function, and the write_to read_to file ().

```
class Person:  1 usage
    """
    A class representing person data.

    Properties:
        first_name (str): The student's first name.
        last_name (str): The student's last name.


    ChangeLog: (Who, When, What)
    O.Richer, 6/06/2025,Created Class
    """
# constructor. so what is that? In a nutshell it is a way to create objects with some initial set of values
# initially my person has no first_name, no Last_name. and with the idea of Constructor we can pass those in
# as an argument to our object itself. so init for "initialization".
    def __init__(self,first_name: str="", last_name: str=""):
        self.first_name = first_name
        self.last_name = last_name
```

Fig1

```
#Properties are a way to get and set instances variables and allow us to put an extra type checking or data validation
# or just extra logic whenever a class is trying to access instances variables.
# ._ is a "notation" perhaps we should call it a "code" that tells python, there are meant to be secret variables and
#( in a perfect world) no one outside of this class is supposed to get access to them.
    @property  6 usages (2 dynamic)
    def first_name(self):
        return self.__first_name.title()
# The extra benefits of properties is that you can also put data validation on this. that means that whenever someone
#tries to set a value here it is going to call this method instead (  .setter)
```

Fig2

```
    @first_name.setter  5 usages (2 dynamic)
    def first_name(self, value: str):
     if value.isalpha() or value == "":
        self.__first_name = value
     else:
        raise ValueError("First name can only have alphabetic character .")
# the last_name deal is the same as above. Just a repeat.
    @property  6 usages (2 dynamic)
    def last_name(self):
        return self.__last_name.title()

    @last_name.setter  5 usages (2 dynamic)
    def last_name(self, value: str):
        if value.isalpha() or value =="" :
            self.__last_name = value
        else:
            raise ValueError("Last name can only have alphabetic character ")
# The "__str__" method return a string representation of an object. here the first name, and the last name of a person.
# In other words, the string method converts the data inside our data class ( student) into a string.
    def __str__(self):
        return f"{self.first_name},{self.last_name}"
```

Fig3

```
# TODO Create a Student class the inherits from the Person class
# TODO call to the Person constructor and pass it the first_name and last_name data
# TODO add a assignment to the course_name property using the course_name parameter
# TODO add the getter for course_name
# TODO add the setter for course_name
# TODO Override the __str__() method to return the Student data
# Inheritance! what is it? while one could probably take the work of this prolific french writer from the 19th century
# in which Balzac wrote about the growing wealth divide in europe in the 19th century and still is to this time and
# explain the idea.
# we will stay in the computer science sphere.
# Inheritance  is this idea that a class can be extended upon to have multiple implementations.
# Inheritance allows us to create a base class ( something generic). In our case the class Person, and it allows us to
#uniquely extend that class to add additional information. In our case, the additional information will be the class
# taken by the student.
# in brief, the class student is going to inherit  everything that comes from the class person ( first name,last name)
# and so in order to use that "inheritance" properly. One still need to go back to that idea on constructor.
# Meaning one needs to find a way to initialize the information properly.
# well, we will call " super constructor".
```

Fig4

```
class Student(Person):  2 usages
    """

    class student inherit from the class person , class student represent a student data
    Properties:
        first_name (str): The student's first name.
        last_name (str): The student's last name.
        course_name (str): The course taken by the student.

    ChangeLog: (Who, When, What)
    O.Richer, 6/06/2025,Created Class

    """
```

Fig5

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

    @property  4 usages (2 dynamic)
    def course_name(self):
        return self.__course_name.title()

    @course_name.setter  3 usages (2 dynamic)
    def course_name(self, value: str):
        if value:
            self.__course_name = value
        else:
            raise ValueError("please enter Course name .")
# Override the Person __str__() method behavior to return a coma-separated string of data
    def __str__(self):
        return f"{self.first_name},{self.last_name},{self.course_name}"
```

Fig6

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

    @property  4 usages (2 dynamic)
    def course_name(self):
        return self.__course_name.title()

    @course_name.setter  3 usages (2 dynamic)
    def course_name(self, value: str):
        if value:
            self.__course_name = value
        else:
            raise ValueError("please enter Course name .")
# Override the Person __str__() method behavior to return a coma-separated string of data
    def __str__(self):
        return f"{self.first_name},{self.last_name},{self.course_name}"
```

Fig7

Here I am going to try to loosely explain Randal explanation and insight pertaining to this “business” of conversion. If we are working a **custom class,** not a built in/pre-made class that comes from python we will have to deal with some kind of trade off. Here we have our own custom student class. We will make up a student object instead of a dictionary object or list object. Student objects contain all the data (first name, last name, and course taken), and we want to store it in a json file. Unfortunately, we cannot work directly with a student object, because the dump command is looking for a dictionary. If multiple rows it will accept a list of dictionary rows.
In brief, we take that object data, convert it into a dictionary format, and then at that point we can open the file, dump the data into the file, and then close that connection. And when we are reading the data back out of the file, we will call the load method. The load method will return to that dictionary. Lastly, since we are using dump and load both work with a list of dictionaries row and working with our own custom collection of objects we have to do that conversion.

Fig8

```
        file = open(file_name, 'w')
        json.dump(student_data_dictionary, file, indent=2)
        file.close()
        print()
        print("Here is the data you just saved!:")
        IO.output_student_and_course_names(student_data=student_data)
    except Exception as e:
        message = "Error: There was a problem with writing to the file.\n"
        message += "Please check that the file is not open by another program."
        IO.output_error_messages(message=message, error=e)
    finally:
        if not file.closed:
            file.close()
```

Fig9

```
        file = open(file_name, 'w')
        json.dump(student_data_dictionary, file, indent=2)
        file.close()
        print()
        print("Here is the data you just saved!:")
        IO.output_student_and_course_names(student_data=student_data)
    except Exception as e:
        message = "Error: There was a problem with writing to the file.\n"
        message += "Please check that the file is not open by another program."
        IO.output_error_messages(message=message, error=e)
    finally:
        if not file.closed:
            file.close()
```

Fig10

1. Running/Executing the script.

I tried to run different scenarios. From the well behave case to cases where the user input does not behave according to the menu.

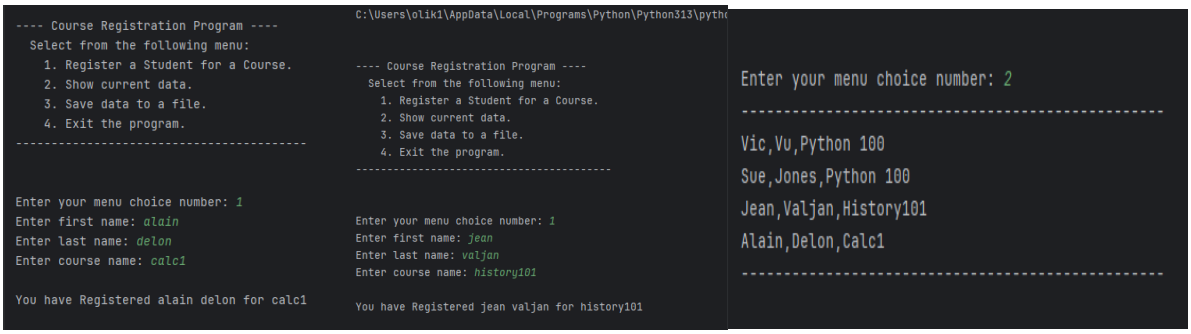

Fig11

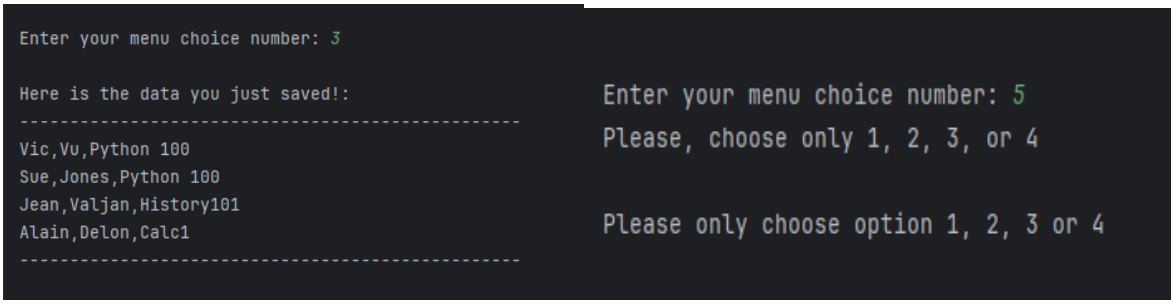

Fig12

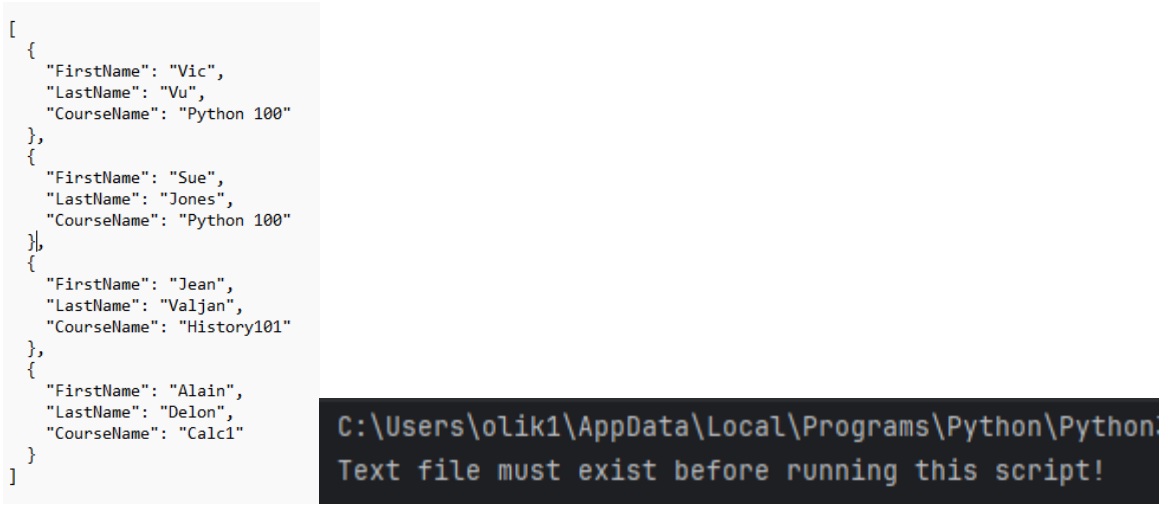

Fig13

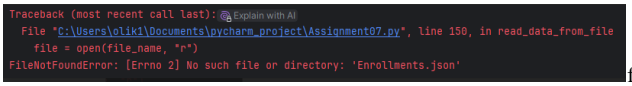

fig14.

Conclusion.

Quite a bit to unpack. Working with classes, constructors, properties, and inheritance was fun. The conversion is not so much fun which brings the fact that in engineering design there is always a trade off, and here the use of the load and dump function came at a cost. For some reason some of my logic for composite names while working fine in previous assignment but were not working at this time. Also, it appears that some validation data are “isolated” and/or delay. I struggled literally an hour on one line of code because of an indentation error. I finally gave up, and put that line as a comment, and “magically” the program works. Finally, on the Lex Friedman podcast, Mr. Guido van Rossum (the creator of the Python Language) was asked if he ever considered brackets instead of indentations. The answer was yes. Too bad he did not follow up on that.