David Goldberg

May 28, 2024

Foundations of Programming: Python

Assignment 07

Link to Github: https://github.com/david-goldberg26/IntroToProg-Python-Mod07.git

Constructors, Properties, and Inheritance

Introduction

In this week's module, we learned how to add on to the usage of classes and make them more complex and versatile. The first topic we discussed was creating a blueprint for more "real world" problems such as creating instances in our code that call back from the class object. We also discussed constructors, properties, setters, and inheritance. These topics are all important for enhancing our code and allow us to create scripts for a larger umbrella of topics.

Classes and Constructors

In this module, we enhanced our knowledge of classes and how to integrate them in our main script section to make our code more efficient. A way we implemented this change was to create a class that can substitute the use of other variables inside other class methods. This calls the introduction for constructors, which is a special method that is used to initialiw the newly created object by setting initial values for its attributes. The method is called '__innit__' and it takes in the parameters that provide the initial values for the object's properties. An example of this is shown below

```
24  class Student:
25      first_name: str
26      last_name: str
27
28      def __init__(self, first_name:str,last_name:str,gpa:float):
29          self.first_name = first_name
30          self.last_name = last_name
31          self.course_name = course_name
```

Figure 1: Use of Constructors

The use of the constructor on line 28 allows the script to pass the initial values from the class 'Student'. When someone tries to create another instance of student, such as reregistering a student for a course, this '__innit__ method will be classed. The 'self' parameter refers to the instance being created and it is always the first parameter in the constructor. It is also knows as an instance method. However, working with classes gives you an extra type of safety net, but JSON files do not have the same functionality. Now it is necessary to create the dictionary before writing to the file. This is shown below.

```
37      def write_data_to_file(file_name: str, student_data: list):
38          try:
39              list_of_dictionary_data: list = []
40              for student in student_data:
41                  student_json: dict \
42                      = {"FirstName": student.first_name, "LastName": student.last_name}
43                  list_of_dictionary_data.append(student_json)
44
45              file = open(file_name, "w")
46              json.dump(list_of_dictionary_data, file)
47              # json.dump(student_data, file)
48              file.close()
```

Figure 2: Classes and JSON Files

Instead of using the '.dump' function alone, you now need to create a dictionary that will get written to the JSON file.

Properties

The use of properties is way to control the access to an object's attributes. For example, if you input data that is "secret" to the user, you would want to use properties to control the input's access. You can use properties to define methods that are known as getters and setters, which work as data validation. The '@property' function or decodor can be used to define the properties in a class. The getter retrieves the value of an attribute, in our case being the students first name and last name, and the setter will allow you to set the value of an attribute and can include validation for the value as well.

```
26  class Student:
27
28      def __init__(self, first_name: str = "", last_name: str = ""):
29          self.first_name = first_name  # set the attribute using the property to provide validation
30          self.last_name = last_name  # set the attribute using the property to provide validation
31
32      @property  # (Use this decorator for the getter or accessor)
33      def first_name(self):
34          return self.__first_name.title()  # Optional formatting code, .upper(will be all uppercase)
35
```

```
36      @first_name.setter
37      def first_name(self, value: str):
38          if value.isalpha() or value == "":  # Optional validation code
39              self.__first_name = value
40          else:
41              raise ValueError("The first name should not contain numbers.")
```

Figure 3: Properties

As seen in the figure above, the getter method is used after the constructor, and it uses the parameter 'self'. To make sure that this variable has limited access, there is an extra underscore after '.self' on line 34. This tells Python that no one outside of the class is supposed to have

access to this item. It is important to know that when creating the property method, its name is supposed to mirror what your variable is. Once the getter is initialized, it is important to include the setter if necessary. In this example we want to use the setter as validation for the variable in the getter. The setter will check if the variable is all alphabetical or an empty string or else it will raise a value error. Overall, properties allow us to clean up a lot of the extra logic done around objects.

Inheritance

Once the ideas of properties and constructors are understood, we can continue onto inheritance. Inheritance is an important concept when dealing with object-oriented programming. It allows a class to inherit attributes and methods from another class. This second class can inherit all the information from the pervious class which promotes code reuse and logical hierarchy. It allows new classes to be built upon existing ones.

```
25
26  class Person:  # TODO Create a Person Class and add the first and last name propertie
27
28      def __init__(self, first_name: str = "", last_name: str = ""):
29          self.first_name = first_name
30          self.last_name = last_name
31
32      @property
33      def first_name(self):
34          return self.__first_name.title()
35
36      @first_name.setter
37      def first_name(self, value: str):
38          if value.isalpha() or value == "":
39              self.__first_name = value
40          else:
41              raise ValueError("The first name should not contain numbers.")
42
43      @property
44      def last_name(self):
45          return self.__last_name.title()
46
47      @last_name.setter
48      def last_name(self, value: str):
49          if value.isalpha() or value == "":
50              self.__last_name = value
51          else:
52              raise ValueError("The last name should not contain numbers.")
53
54      # TODO Override the __str__() method (Done)
55      def __str__(self):
56          return f"{self.first_name},{self.last_name}"
57
```

```
59  class Student(Person):
60
61      def __init__(self, first_name: str = "", last_name: str = "", gpa: float = 0.0):
62          super().__init__(first_name=first_name,last_name=last_name)
```

Figure 4: Inheritance

From the example above, we see a class called Person, which contains a constructor, getters and setters, as well as a second class called Student, which inherits the class person, shown on line 61. The constructor used, once Student inherits Person, uses the super class, super(). This is used

so when the student class is being constructed, it will also construct the person class as well. All the initialization happens in the the class Person will also occur in the Student class. This allows you to not have to repeat getters and setters in the class, you only need to add additional getters and setters that are unique to the Student class.

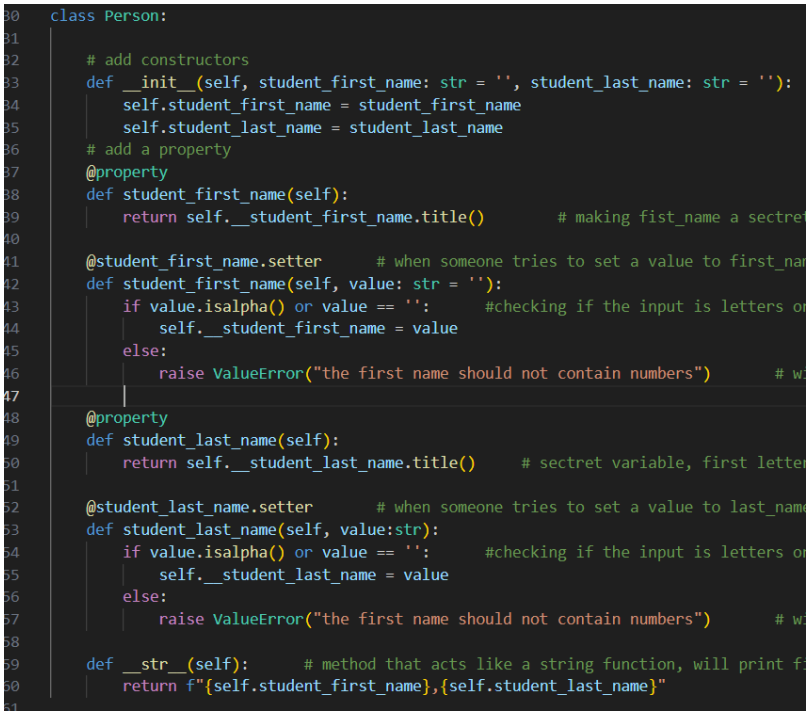
Writing the Script

In this week's module we introduce the use of constructors, properties with setters and getters, and inheritance in our ongoing student registration example. In this script, there will be a new class introduced called Person and the script will have the original class, Student, which will inherit the attributes from the Person class. First, we need to define constants and variables we will be using in the main section of the script, which will be the student menu, the menu choice, and the list of student inputs. This is done before any objects are defined.

```
10  import json
11  from typing import TextIO
12
13  # Define the Data Constants
14  MENU: str = '''
15  ---- Course Registration Program ----
16    Select from the following menu:
17      1. Register a Student for a Course.
18      2. Show current data.
19      3. Save data to a file.
20      4. Exit the program.
21  -----------------------------------------
22  '''
23  # Define the Data Constants
24  # FILE_NAME: str = "Enrollments.csv"
25  FILE_NAME: str = "Enrollments.json"
26
27  # Define the Variables
28  students: list = []  # a table of student data
29  menu_choice: str  # Hold the choice made by the user.
```

Figure 5: Initialization of variables and constants

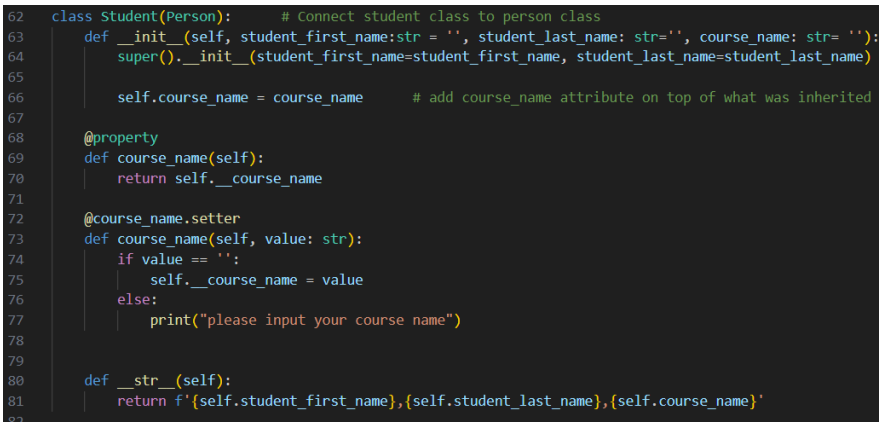
Once the constants and variables are defined, we can create our first object called Person, which will hold the constructor and attributes for the student's first name and student's last name. This class will also contain getters and setters that will make sure the student's names are all letters and are not left empty during the input process. The Person class is shown below.



```
class Person:

    # add constructors
    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
    # add a property
    @property
    def student_first_name(self):
        return self.__student_first_name.title()        # making fist_name a sectret

    @student_first_name.setter      # when someone tries to set a value to first_nam
    def student_first_name(self, value: str = ''):
        if value.isalpha() or value == '':      #checking if the input is letters or
            self.__student_first_name = value
        else:
            raise ValueError("the first name should not contain numbers")       # wi

    @property
    def student_last_name(self):
        return self.__student_last_name.title()     # sectret variable, first letter

    @student_last_name.setter       # when someone tries to set a value to last_name
    def student_last_name(self, value:str):
        if value.isalpha() or value == '':      #checking if the input is letters or
            self.__student_last_name = value
        else:
            raise ValueError("the first name should not contain numbers")       # wi

    def __str__(self):      # method that acts like a string function, will print fi
        return f"{self.student_first_name},{self.student_last_name}"
```

Figure 6: Person Class

As seen on line 33, the constructor is created for the Person class which contains the attributes ‘student_first_name’ and ‘student_last_name’. We can use properties for each instance of the students first and last name to make sure that no one outside this class has access to the variables. The getter and setter for student_first_name is from line 37 to 46. The getter and setter for student_last_name are from lines 48 to 57 which do the same actions as the getter and setter for first name. The last method of the Person class is the __str__ method which will print the instance of the student’s first and last name. The next class is the Student class, shown below.



```
class Student(Person):      # Connect student class to person class
    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      # add course_name attribute on top of what was inherited

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

    @course_name.setter
    def course_name(self, value: str):
        if value == '':
            self.__course_name = value
        else:
            print("please input your course name")


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

Figure 7: Student Class

Instead of rewriting everything that was included in the Person class, we can just inherit the Person class, which is done in line 62. There is also a new constructor made for the addition of the student’s course name, but since this class is inheriting attributes, it uses ‘super()’ on line 64

which will inherit the instances of student_first_name and student_last_name into the class. For the new attribute course_name, it will have a setter and getter which for each instance if it is empty. If so it will output a message to the user. It will also use the __str__ method which will print the instances of student_first_name.

There are two other classes used in this script, 'FileProcessor' and 'IO', which were explained in the previous assignment. But since there were a few changes to those classes I will explain the updates.

```
87      @staticmethod
88      def read_data_from_file(file_name:str, student_data: list):
89          '''
90          Function that reads in data from a JSON file and then into a dictionary
91          :param file_name: A string indicating the file name
92          :param student_data: dictionary from students inputs
93          :return: list of student data
94
95          '''
96          file: TextIO = None
97          try:
98              list_of_dictionary_data: list = []  # classes change the functionality of json, need to create a dictionary
99              file = open(file_name, "r")     # open the file in read mode
100             list_of_dictionary_data = json.load(file)
101             for student in list_of_dictionary_data:
102                 new_student = Student(student_first_name=student["FirstName"], student_last_name=student["LastName"], course_name=student["CourseName"])
103                 student_data.append(new_student)
104                     # load the previous dictionaries from the json file
105             #print(student_data)
106             file.close()
107             print("data retreived")
108         except Exception as e:
109             IO.output_error_message(message='text file not found\n', error = e)     # calling IO class for structured error
110         finally:
111             if file == False:
112                 file.close()
113         return student_data
```

Figure 8: Reading from JSON

Shown above is the read_data_from_file which reads in the existing information from the JSON file. Since we are doing object oriented programming now the simple use of 'json.load' will not work as it did before. This means we need to create the dictionary once the information has been read from the file. In order to do this we need to add another parameter which will call the Student class. This allows us to input the existing student's first name, last name, and course name to the associated key shown on line 103. Once everything is written to 'new_student' the instance will get appended to student_data which will get returned. The same changes need to be applied to writing the JSON file, which is shown below.

```
116     @staticmethod
117     def write_data_to_file(file_name:str, student_data:list):
118         '''
119         Function that reads the students dictionarties into the JSON File
120         :param file_name: A string indicating the file name
121         :param student_data: dictionary from students inputs
122
123         '''
124         try:
125             list_of_dictionary_data = []
126             for student in students:
127                 student_json: dict \
128                 = {"FirstName": student.student_first_name, "LastName":student.student_last_name, "CourseName":student.course_name}
129                 list_of_dictionary_data.append(student_json)
130
131             file = open(file_name, "w")     # open a file to write to
132             json.dump(list_of_dictionary_data, file)       # write the dictionary/dictionaries to the json file
133             file.close()
134             print("The following data was saved to file!\n")
135             IO.output_student_courses(student_data=student_data)        # calling IO class to output inputs
136         except Exception as e:      # Structured error handling if any exceptions occur
137             if file.closed == False:
138                 file.close()
139             IO.output_error_message(message='There was a problem with writing the file', error=e)       # calling IO class for struc
140         finally:
141             if file.closed == False:
142                 file.close()
```

Figure 9: Writing to JSON

Before we can use the 'json.dump' command, it is necessary to create the dictionary that will be dumped to the JSON file. Using the keys established from the function before, I can create the dictionary shown in line 128. Once the dictionary is build I can 'dump' it to the JSON file. The next change that was made was in the input_student_data function, where the student is asked to input.

```
194     @staticmethod
195     def input_student_data(student_data:list):
196         '''
197         Function allows for students to input their names and courses
198          :param student_data: list of dictionaries that are filled by students inputs
199         :return: dictionary (list)
200           '''
201         try:
202             new_student = Student()
203             new_student.student_first_name = input("Enter the student's first name: ")      # student will input first name
204             new_student.student_last_name = input("Enter the student's last name: ")      # student will input first name
205             new_student.course_name = input("Please enter the name of the course: ")
206             student_data.append(new_student)
207             print(f"You have registered {new_student.student_first_name} {new_student.student_last_name} for {new_student.course_name}.")
208         except ValueError as e:
209             IO.output_error_message(message= "inputted name is not the correct data type", error=e)     # calling IO class for structured
210         except Exception as e:
211             IO.output_error_message(message= "Error: Problem with entering your data", error=e)     # calling IO class for structured erro
212         return student_data
213
```

Figure 10: Inputting student data

The changes made in this function are calling the Student class and its attributes. Now when the student inputs their name they will associate the input with the class's attribute for student_first_name, student_last_name, and course_name. Once all the values are inputted, the list will get appended to the larger list of lists 'student_data'. The same goes for outputting the data to the user.

```
214     @staticmethod
215     def output_student_courses(student_data:list = []):
216         '''
217         Function is to output the students compelete dictionary
218         :param student_data: list of students inputs
219         '''
220         print("-" * 50)
221         for student in student_data:
222             print(f'student {student.student_first_name} '
223                   f'{student.student_last_name} is enrolled in {student.course_name}')
224         print("-" * 50)
```

Figure 11: outputting the data

In the figure above, this function will loop through each instance in student_data and print the corresponding class attribute out to the user. The main portion of the script stays the same as the previous week's assignment, but I will include is in here as well.

```
students = FileProcessor.read_data_from_file(file_name=FILE_NAME, student_data=students)

while True:

    IO.output_menu(menu=MENU)       # calls IO class and a method to output the menu
    menu_choice = IO.input_menu_choice()        # calls IO class and a method to get student

    if menu_choice == "1":
        students = IO.input_student_data(students)      # if choice is 1 it will call the in
        continue

    elif menu_choice == "2":
        IO.output_student_courses(students)     # if choice is 2 it will call the output stu
        continue

    elif menu_choice == '3':
        FileProcessor.write_data_to_file(FILE_NAME, students)       #if choice is 3 it will
        continue

    elif menu_choice == '4':        # break out of the program and exit
        break

print('End of program')
```

Figure 12: main body

With all the classes and functions, the main body of the script is left as simple as it possibly could be. Classes allow for the main body to be extremely fluid and readable for the user. Each choice is associated with a function that has been previously defined in either the FileProcessor Class or the IO class.

Summary

Overall, this module helped me understand the importance of constructors, properties, and inheritance. These topics allow for initializing attributes as well as using properties on those attributes. The properties include setters and getters which affect these attributes that only that class can have access to. Inheritance is extremely important because it eliminates redundancy in the code, which you can see from this assignment. These topics are very useful and I will definitely use them in future programs.