Phiphat Pinyosophon

August 11, 2024

Foundation of Programming: Python

Assignment 07

https://github.com/pinyosophon/python110-Summer2024

# PROPERTIES, CONSTRUCTOR, AND INHERITANCE

## Introduction

In this week assignment, we were to dive deeper in to usage of creating class by modifying given code, part of the assignment is to create properties and use constructor to initialize properties(attributes) for each instance of the class when it's created. We also learned about inheritance, how to create a validation system in our classes, and how to implement all that we learned into our code.

## Properties and Constructor

Properties in Python are used to create managed attribute, allowing you to define methods that control access to instance variable. Properties are often used with constructor, which is used to initialize attribute for each instance of the class when created. For example:

```
28  class Person:
29      # TODO Add first_name and last_name properties to the constructor (Done)
30      def __init__(self, first_name:str = '', last_name:str = ""):
31          '''
32          creating property with constructor for person class with
33          first_name and last_name as parameters to initialize
34          :param first_name: string parameter we assign for first_name
35          :param last_name: string parameter we assign for last_name
36          '''
37          self.first_name = first_name
38          self.last_name = last_name
```

***Figure01: Demonstrate usage of class with property and constructor to initialize attributes***

As you can see in ***Figure01***, In line 28: we created a class called **Person**, and in line 30, **__init__()** method, or constructor automatically is invoked, and it will initialize properties for this class with **first_name**, and **last_name** parameter. "**self**" refers to instance of that class, when a code executes and create object, self will refer to that specific object. In line 37 and 38, **self.first_name** and **self.last_name** are instance variables (or instance

attributes). When an object is created, values of **first_name** and **last_name** parameters will be passed onto their assigned instance variables. So to summarize these are the steps taken during that process:

1. **Object creation:** an object of "Person" class is created

2. **Parameter Passing:** Values for first_name and last_name parameter are pass onto constructor

3. **Instance variable assignment:** Values are assigned to self.first_name, and self.last_name

4. **Object initialization:** Object is now initialized with provided values.

## Getter

We go through all these is so that we can control access and validation of these variables by using getter and setter method, as you can see in this following example:

```
40      # TODO Create a getter and setter for the first_name property (Done)
41      @property  #this part is getter
42      def first_name(self): #retrieve first_name
43          return self.__first_name.title() #return first_name, and make sure it's capitalized
44
45      @first_name.setter #this part is setter
46      def first_name(self, value: str):
47          if value.isalpha() or value == '':
48              self.__first_name = value
49          else:
50              raise ValueError("First name must be letters")
51
```

***Figure02a: using getter and setter method to retrieve and validating data***

In Figure02a, usage of getter and setting in my code is demonstrated. In line 41, "**@property**" decorator is used to turn a method into a getter. Then in following lines, we have:

```
41      @property  #this part is getter
42      def first_name(self): #retrieve first_name
43          return self.__first_name.title() #return first_name, and make sure it's capitalized
```

***Figure02b: using getter to retrieve value***

The following lines after @property are used to retrieve value of instance variable associated with "**first_name**" property. Once it was retrieved, the data "return" as private instance variable called "**__first_name**". (Double underscore, or dunder, indicates that this is intended to be private instance variable.) This private instance then is having its first

letter capitalized by **.title()** function, and are stored in memory to be called upon by setter method.

## Setter

After we use **@property** as getter to retrieve data, we are now ready to set it with "setter". As you can see in **Figure03:**

```
40          # TODO Create a getter and setter for the first_name property (Done)
41          @property  #this part is getter
42          def first_name(self): #retrieve first_name
43              return self.__first_name.title() #return first_name, and make sure it's capitalized
44
45          @first_name.setter #this part is setter
46          def first_name(self, value: str):
47              if value.isalpha() or value == '':
48                  self.__first_name = value
49              else:
50                  raise ValueError("First name must be letters")
```

***Figure03: using setter***

Firstly, the getter and setter's name need to match, as you can see in Figure03, the names with yellow underlines are exactly the same. "**.setter**" is used to tell that this is setter method. We're telling this setter that we are going to assigning an incoming "string" to "value" variable in the parentheses.

```
def first_name(self, value: str):
```

Then, we run a validation for this setter by using if/else. We gave it a condition that, if this incoming data in value variable is alphabetical  or if it is an empty string, set it to this private instance "**__first_name**", else raise a value error and show custom error message that the first name must be letters:

```
if value.isalpha() or value == '':
    self.__first_name = value
else:
    raise ValueError("First name must be letters")
```

I have set up getter and setter for both **first_name** and **last_name** properties of Person class. And then we return those with fstring format like this in **Figure04**.

```
64      # TODO Override the __str__() method to return Person data (Done)
65      def __str__(self):
66          return f'{self.first_name},{self.last_name}'
67
```

***Figure04: using string method to return as formatted string***

We use string method to return the formatted incoming data as string, starting with **first_name**, then **last_name**, with a comma in between. This is to define how this object should be represented, and it is used to integrating with getters we created earlier. This is to make sure that the string representation will include whatever formatting or process defined in getter method. In this case, we tell it to capitalizing the first letter in first_name and last_name with title() function.

## Inheritance

Inheritance is a concept widely used in object-oriented programming, it allows one class to inherit attributes from another class. it promotes reusability and establish a hierarchy between classes. Inheritance will have **Superclass (class that's being inherited from)** and **Subclass (class that inherits from Superclass).** In our previously shown examples, we had created Person class, and this was to be our Superclass. Now we would create Student as our Subclass and would inherit properties from Person class to to use in this assignment.

```
68  # TODO Create a Student class the inherits from the Person class (Done)
69  class Student(Person):
70      # TODO call to the Person constructor and pass it the first_name and last_name data (Done)
71      def __init__(self, first_name:str = '', last_name:str = '', course_name:str =''):
72          super().__init__(first_name = first_name, last_name = last_name)
73          # TODO add a assignment to the course_name property using the course_name parameter (Done)
74          self.course_name = course_name
```

***Figure05a: creating Student Subclass and inherit from Person Superclass***

In Figure05a, we created Student Subclass by putting name of Superclass in parentheses of Subclass as follow in line 69:

```
69  class Student(Person):
```

***Figure05b: line of code we used to create Subclass from Superclass***

Then we need to define the constructors for Student class in line 71:

```
71      def __init__(self, first_name:str = '', last_name:str = '', course_name:str =''):
```

***Figure05c: creating constructors for Student class***

As you can see that there's **first_name** and **last_name** parameters, similarly to Person class. Additionally, there's a **course_name:str** parameter as well.

```
72          super().__init__(first_name = first_name, last_name = last_name)
73          # TODO add a assignment to the course_name property using the course_name parameter
74          self.course_name = course_name
```

***Figure05d: inheriting from Superclass***

To inherit from Superclass, first you must tell Python like this:
**super().__init__(parameter_from_Superclass = parameter_from_Subclass)**

Or as you can see in **Figure05d** at **line 72**, this would tell Python to inherit **first_name** and **last_name** parameters from Person Superclass and assign them to its own parameters. And then to create constructor for **course_name** is pretty much the same as how you created **first_name** and **last_name** for Person class:

```
74          self.course_name = course_name
```

***Figure05e: creating constructor***

## Validating course_name property

Validating **course_name** property was a little different than doing so for **first_name** and **last_name** property, and this is because a name of a course usually include letter, number, and spaces. Requirement for validating **course_name** property is a little more complex than other properties.

```
77      @property
78      def course_name(self):
79          return self.__course_name.title()
80      # TODO add the setter for course_name (Done)
        6 usages (4 dynamic)
81      @course_name.setter
82      def course_name(self, value: str):
83
84          if all(char.isalnum() or char.isspace() for char in value) or value == '':
85              self.__course_name = value
86          else:
87              raise ValueError("course name must be letters")
88
89      # TODO Override the __str__() method to return the Student data (Done)
90      def __str__(self):
91          result = super().__str__()
92          return f'{result},{self.course_name}'
```

***Figure06: Validating course_name***

As opposed to just use .**isalpha()** function as we did when we validated **first_name** and **last_name**, we added more conditions to check for its validation, and in addition to that, we have to check each and every element whether it meets the condition or not. When we created condition for **first_name** and **last_name**, we did in such a way that we expected them to be one single word as first name or last name would have no spaces or numbers in them, therefore **value.isalpha()** met that specification perfectly as it checked the whole word if it was alphabetic or not. However, that isn't the case for a name of a course, for example "Python 100" has both number and space in it.

And to do that, we have to use **all()** function as you can see in ***Figure06*** line 84:

This **all()** function will iterate through each element and check whether it is:

1 .**isalnum()** function, check if each element is an alphabet or number or not.

2. **isspace()** function, check if each element is an empty space or not.

And to use **all()** function, I must also use for loop inside to check for each element. This is a boolean function, and will only **return True if every element passes condition** set by user, in this case, check if it's letters, number, or space. So any special characters like !, #, @, &, etc. will not pass the check and will return False. In addition, **value() == ''** will allow user not putting anything in **course_name** as well.

## Wrapping up Inheritance

And to finish this inheritance process, you must combine the inherited properties from Superclass and the property from Student together like this:

```
89      # TODO Override the __str__() method to return the Student data
90      def __str__(self):
91          result = super().__str__()
92          return f'{result},{self.course_name}'
```

***Figure07: combine inherited properties from Superclass and Subclass together***

This code in line 91 calls string method from Person Superclass, it returns string representation that we set up previously as you see in **Figure04** above. Then we combine it with its own return property in line 92.

## Making use of Validation system

The rest of the code is still not making use of the validation system we had set up in classes, and this is because in the given code, it was set up to manipulate string and dictionary, instead of manipulate instances. As you can see in **Figure08** below, at line 66 data and dictionary from .json file was directly loaded to **student_data:list**. Then it was return to be used later in other function in line 74.

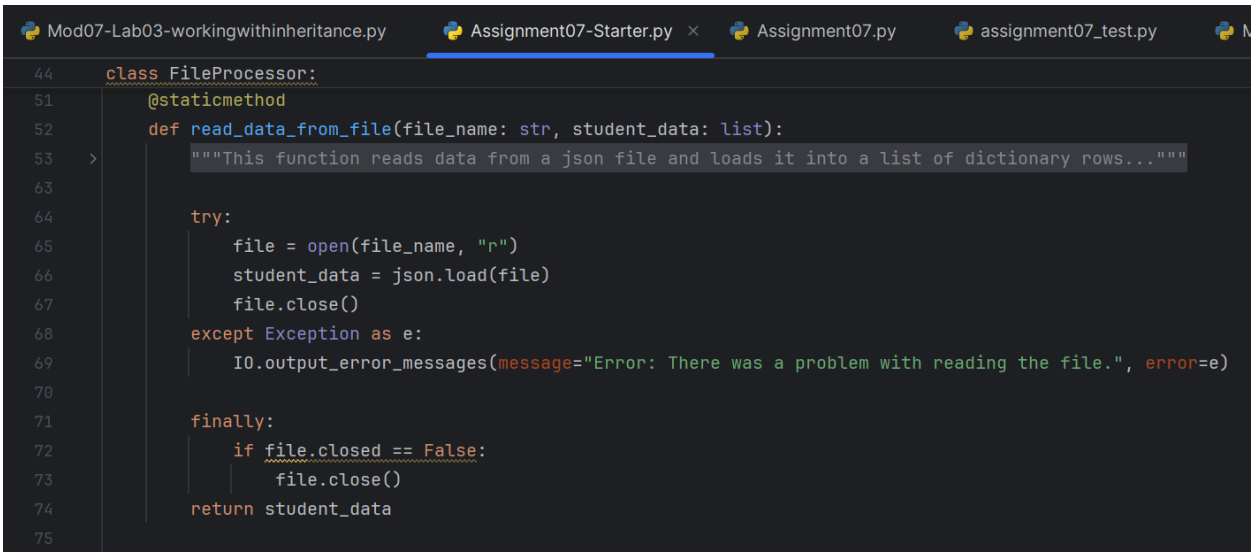


***Figure08: Code from starter file where .json is loaded directly to student_data***

In order to use our validation system, we must first map the parameters we set up to dictionary from json file, like this example in **Figure09:**

```python
class FileProcessor:
    """A collection of processing layer functions that work with Json files..."""
    @staticmethod
    def read_data_from_file(file_name: str, student_data: list):
        """This function reads data from a json file and loads it into a list of dictionary rows..."""

        try:
            file = open(file_name, "r")
            list_of_dict_data = json.load(file) #store list of dictionary from json to here
            for student in list_of_dict_data:
                student_obj: Student = Student(first_name = student["FirstName"],
                                               last_name = student["LastName"],
                                               course_name = student["CourseName"])
                student_data.append(student_obj)
            file.close()
        except Exception as e:
            IO.output_error_messages(message="Error: There was a problem with reading the file.", error=e)

        finally:
            if file.closed == False:
                file.close()
        return student_data
```

***Figure09: updated code by mapping json dictionary to parameters from constructor***

In **Figure09**, when .json is loaded, we stored the loaded data into **list_of_dict_data**, which we used to go through them in for loop. In for loop, we created **student_obj** for each set of dictionary in .json, once it was created, constructor that we set up for Student class is invoked, and created 3 parameters we initially set up, which were **first_name, last_name,** and **course_name.** For each parameter, it mapped related dictionary to each of them. And once that process was finished, it then appended to **student_data** list. That **student_data** list was returned, and now ready to be used by other functions in the code.

## Updating the rest of the code

Now that we set up the code to create object instance from Student class, map parameters to dictionary from .json file, we had to update the rest of our code to use reference from Student object instances instead of reading it from dictionary.

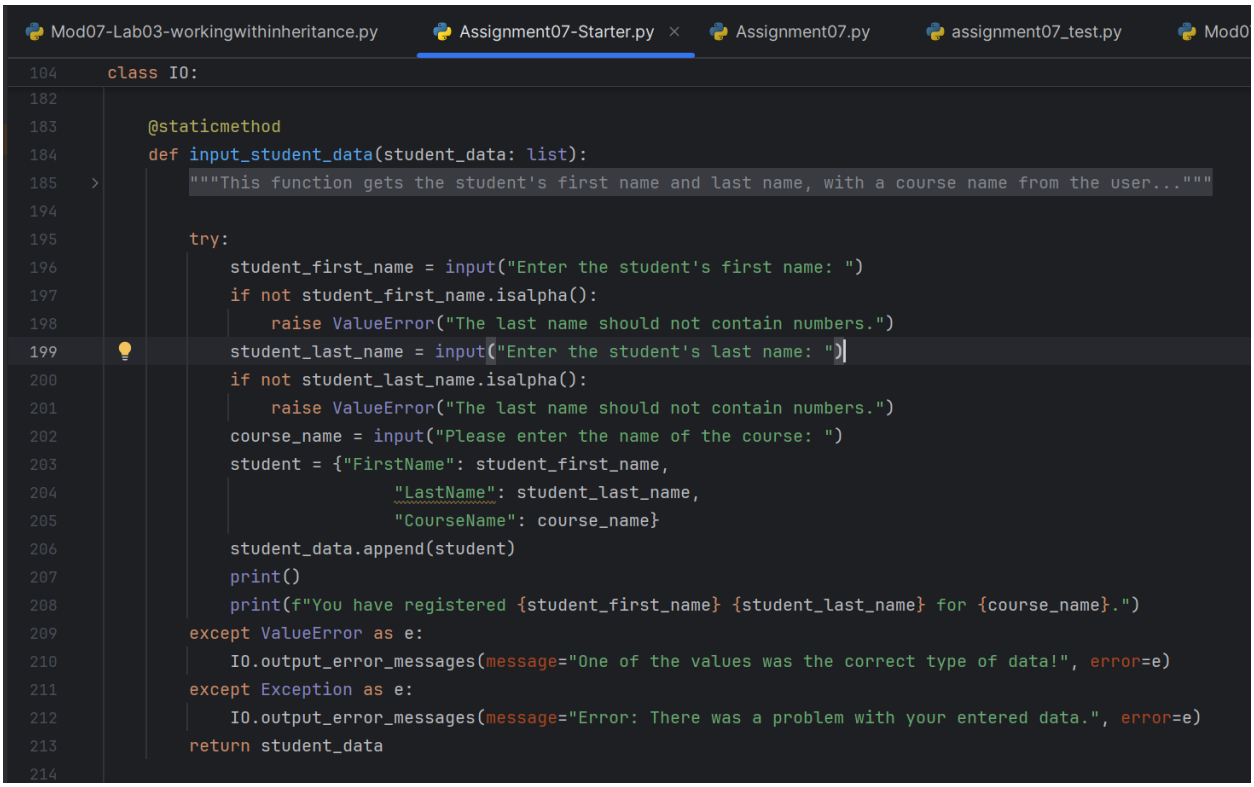


***Figure10a: what the code look like originally in the given file***

In the starter file we were given, in **Figure10a**, you can see that the validation system is done here. It will raise **ValueError** for **first_name** and **last_name** if it detect any input that isn't an alphabet. This version of the code also used 3 local variables created specifically for this function. In comparison, **in Figure10b** below, was what our code currently look like.

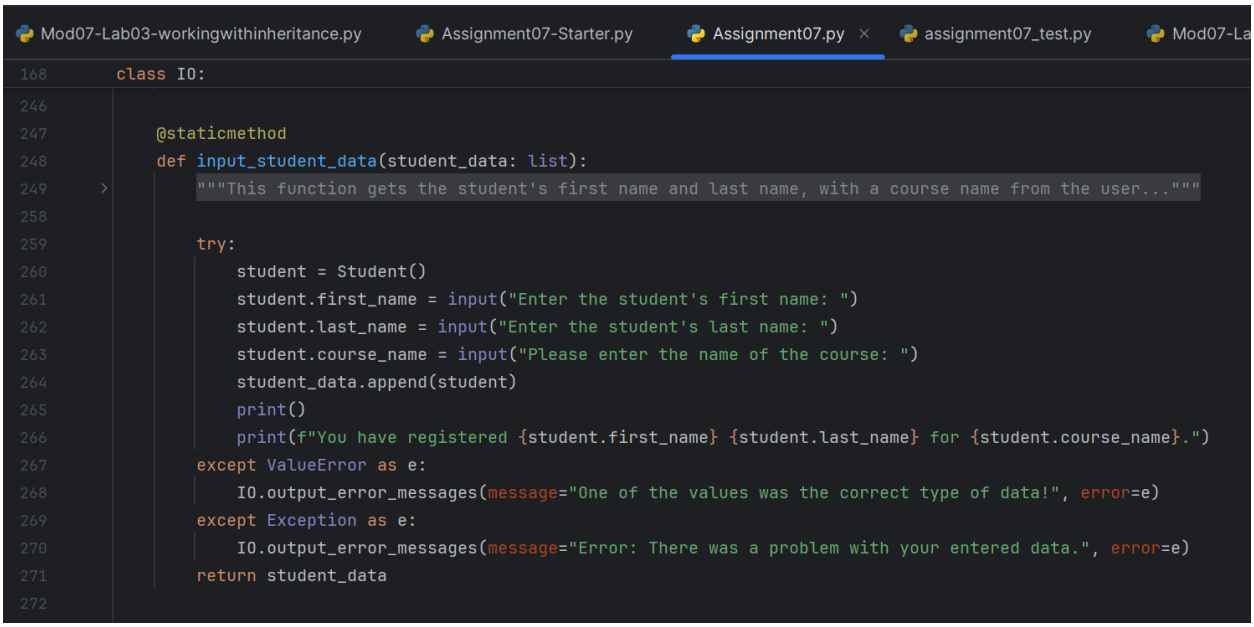


***Figure10b: Updating this function by using Student class as object instance***

In this function, we made use of validation system we had created in **Person class** and Student class. And instead of using local variables, we referenced them to object instances we created instead. At line 260, this code created object instances from **Student class** with **student = Student()**. Then in line 261 to 263, it mapped each input from user to its related parameter, **first_name, last_name,** and **course_name**, by using **student.first_name, student.last_name**, and **student.course_name** respectively. Once completed, it appended all that to **student_data**. It still printed out the information we needed by calling on those instances with the same **student.first_name, student.last_name**, and **student.course_name.**

This similar changes were done to other function as well as you can see in **Figure11a** and **Figure11b**

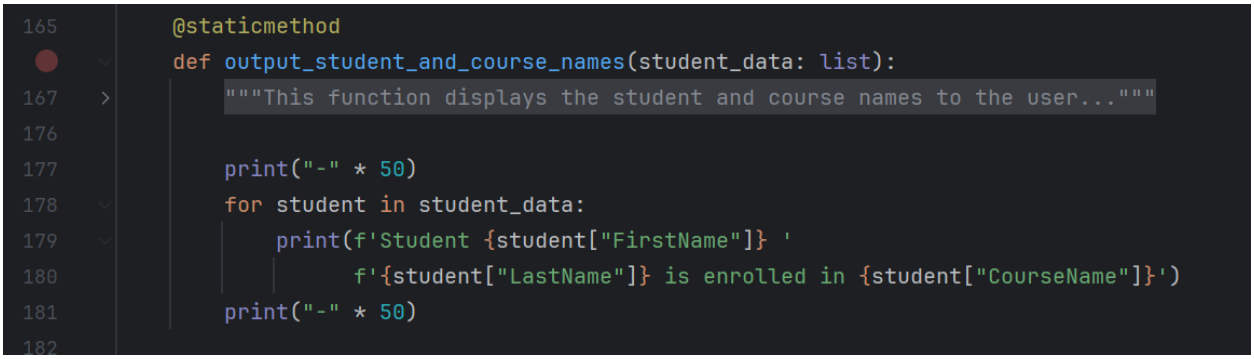


***Figure11a: what the code look like originally in the given file***

```
229     @staticmethod
230     def output_student_and_course_names(student_data: list):
231         """This function displays the student and course names to the user..."""
240
241         print("-" * 50)
242         for student in student_data:
243             print(f'Student {student.first_name} '
244                   f'{student.last_name} is enrolled in {student.course_name}')
245         print("-" * 50)
```

***Figure11b: Updating this function by using Student class as object instance***

Lastly, we needed to update the process of writing the file out to .json. We could do so like this example from **Figure12** below:

```
134     @staticmethod
135     def write_data_to_file(file_name: str, student_data: list):
136         """This function writes data to a json file with data from a list of dictionary rows..."""
146
147         try:
148             list_of_dict_data: list = []
149             for student in student_data:
150                 student_json: dict = {"FirstName": student.first_name,
151                                       "LastName": student.last_name,
152                                       "CourseName": student.course_name}
153                 list_of_dict_data.append(student_json)
154             file = open(file_name, "w")
155             json.dump(list_of_dict_data, file)
156             file.close()
157             IO.output_student_and_course_names(student_data=student_data)
158         except Exception as e:
159             message = "Error: There was a problem with writing to the file.\n"
160             message += "Please check that the file is not open by another program."
161             IO.output_error_messages(message=message, error=e)
162         finally:
163             if file.closed == False:
164                 file.close()
```

***Figure12: Writing to json file by mapping value from each property to dictionary key***

In this function, at line 148, we created a list called **list_of_dict_data** to use as temporary storage to hold dictionary created from each Student object instances. From line 149 to 152, we used for loop to iterate through each object in **student_data**, and creating dictionary for each of the object in **student_data**. In order for this to work correctly, we need to map correct attribute from each instance to its appropriate key in the dictionary. Then we stored that dictionary data in **student_json** variable. Once it finished going through every iteration, data in **student_json** would be appended to **list_of_dict_data** at line 153, and lastly, it was written to .json file in line 155.

## Summary

The process of creating property and constructor for class to use getter and setter and using inheritance, while making code becoming more complex, but it allows for modification and validation easily because we are now referencing object instances instead of modifying each of the string in each function. In a long run, it shorten our code and changes become easily done as you only have to do it once during getter/setter where you are doing validation.