Patrick Moynihan
May 23, 2024
Course: Foundations of Programming: Python
Assignment: 07
GitHub link: https://github.com/Pyronious/IntroToProg-Python-Mod07

Exercise: Data Classes and Subclasses

Contents

Introduction

The goal of this exercise is to learn about working with objects, properties, and subclasses. The functionality of our program is the same as last week. We're refactoring the code so that it uses new classes named **Person** and **Student**. We'll use objects derived from the **Student** class to hold our student data instead of dictionaries like we did last week. We'll also learn about subclasses, which can inherit features from their parent class.

Data Preparation

Our first step is to define the constants and variables we will be using throughout the script.

Constants

We start by declaring our constants. Nothing new this week.

```
MENU: str = '''
------ Course Registration Program ------
  Select from the following menu:
    1. Register a Student for a Course.
    2. Show current data.
    3. Save data to a file.
    4. Exit the program.
-----------------------------------------
'''
FILE_NAME: str = "Enrollments.json"
KEYS: list = ["FirstName", "LastName", "CourseName"]

# Define the global data variables
menu_choice: str = ''  # Hold the choice made by the user.
students: list = []  # List of data for all students
saved: bool = True  # Tracks whether newly added data has been saved
```

Figure 1: Declaring constants.

Variables

Nothing new this week, just these three:

```
# Define the global data variables
menu_choice: str = ''  # Hold the choice made by the user.
students: list = []  # List of data for all students
saved: bool = True  # Tracks whether newly added data has been saved
```

Figure 2: Declaring global variables.

Defining Classes

We're expanding our understanding of classes this week to include classes that contain properties. Properties work kind of like variables, but with added functionality. We can use classes with properties to create our own custom objects which contain both properties (data) and methods (functions). Each object instance has its own set of properties that we can set or get.

This week we'll be adding two new classes to our code: **Person** and **Student**. The **Student** class will derive from the **Person** class, which means that **Student** will inherit all the properties and methods of **Person** (it's parent class).

Person

We'll start with our first new class, **Person**. We'll declare it and give it a doc string:

```
class Person:
    """
    Class for storing information about a person

    ChangeLog:
        Patrick Moynihan, 2024-05-22: Created class
```

Figure 3: Declaring the Person class.

Nothing new here so far.

Init

Now we'll add our initializer method, which uses the keyword **__init__** as its method name.

```
    def __init__(self, first_name: str = "", last_name: str = ""):
        """
        Initialise a new Person object

        :param first_name: First name of the person
        :param last_name: Last name of the person
        """
        self.first_name = first_name
        self.last_name = last_name
```

Figure 4: The initializer method for our Person class.

The initializer automatically runs any time a new object of type **Person** is created. It defines what attributes exist inside the object, and what their default values are. Like any other method, you can pass arguments to the parameters **first_name** and **last_name** when you create a new object.

Line 52 assigns the attribute **self.first_name** the value that was passed to the arguement **first_name**. Line 53 does the same, but for last_name.

Property Getters

Data objects typically store their data in private attributes (called **properties**), which can't be modified directly by any code outside of the object's class. This is called encapsulation. This means we must have a special method called a getter to allow external code to retrieve data from an object's properties. We do this using property a "getter" (formally referred to as an "accessor").

```
69      @property
70      def first_name(self) -> str:
71          """
72          Gets the person's first name
73
74          Returns:
75              A string containing the first name of the person
76          """
77          return self.__first_name  # Returns the private attribute
```

Figure 5: The getter for the first_name property.

In Figure 5 we see code that creates the getter method for **first_name**. The **@property** decorator means this method will automatically be used as a "getter" when external code needs to get the value of **first_name** from the object. **self** is used to indicate that we are talking about the **first_name** attribute that belongs to the object itself.

Line 77 simply returns the value of the private attribute self.__first_name. Private attributes are designated in Python by a double-underline prefix as seen here.

For example, if you have an object of type **Person** named **resident**, you could get the resident's first name with something like **x = resident.first_name**. The getter function would then return the value of the private attribute and it would be assigned to the variable **x**. Note that you don't have to use resident.firstname() (with the parentheses) like you normally would with a method. That's because the **@property** decorator makes this method "magically" behave like a simple public attribute accessor.

Property Setters

A setter (formal name “mutator”) is like a getter but it puts values into an object’s properties. It’s useful because we can check any data that’s passed in to be stored, validate it, and then store it in the property.

```
79      @first_name.setter
80      def first_name(self, value: str):
81          """
82          Sets the person's first name
83          """
84          # Validate incoming data
85          try:
86              if Student.validate_name(value):  # Use our custom validation method to check the name
87                  self.__first_name = value.title()  # Store private attribute
88              else:
89                  raise ValueError(f">>> First name must use only letters.\n")
90          except ValueError as e:
91              IO.output_error_messages(e)
```

Figure 6: The setter for the first_name property.

The **@first_name.setter** decorator on line 79 is used to define this method as a setter for the property **first_name**. The method on line 80 has two parameters, **self** and **value**. Again, **self** indicates we want to work with the **first_name** attribute that belongs to the object itself. The **value** attribute accepts a string that is passed in to the setter. We use **value** in the validation code (lines 85-91) to see if it’s a good or bad value. If it is good, line 87 stores the value in the private attribute **self.__first_name**.

Validation

In Figure 6 we see some data validation happening in line 86. We take the value that was passed in to the setter, and hand it off to Student.validate_name() to check if it is good value or a bad value. Here’s what the validator does:

```
@staticmethod
def validate_name(name: str):
    """
    Validates a name to ensure it contains only alpha or space characters.

    :param name: Name to be validated
    :return: Returns True if the name is valid, otherwise False
    """
    # If the name contains only alpha characters (ignoring spaces), or is empty, validate it as good.
    if name.replace(' ', '').isalpha() or name == '':
        return True
    else:
        return False  # Data validation failed
```

Figure 7: The validate_name method.

This method is used to check if the name the user entered is valid. We consider a valid name to be one that contains only alphabetical characters or spaces, or an empty string. Line 64 checks **name** to see if it only contains alpha characters using the **.isAlpha()** method. Before we hand the name off to **isAlpha()**, we strip the spaces from the name using **.replace(' ', '')**. That removes spaces prior to the alpha check, because some names have spaces in them ("del Toro"), but **isAlpha()** normally rejects spaces.

Wrapping up the Person class

We'll wrap up the rest of our Person class by adding a **last_name** property setter and getter. It's the same as the **first_name**, so no need to go over it again.

Student

We're going to derive our **Student** class from our **Person** class. That means we'll inherit all the work we did on **Person** including all the properties (**first_name** and **last_name**) and methods (**validate_name()**).

```
118 class Student(Person):
119     """
120         Subclass of Person for storing information about a student
121 
122         ChangeLog:
123             Patrick Moynihan, 2024-05-22: Created class
124     """
```

Figure 8: Creating the Student subclass.

Line 118 declares our class using the **class Student(Person)** notation. This means we want to create a new class name **Student**, using **Person** as our parent (or "super") class. We inherit all the features of our parent class when we create a subclass in this way.

```
    def __init__(self, first_name: str = "", last_name: str = "", course_name: str = ""):
        """
        Initialize a new Student object
        :param first_name: First name of the student
        :param last_name: Last name of the student
        :param course_name: Course name the student is registered for
        """
        super().__init__(first_name, last_name)
        self.course_name = course_name
```

Figure 9: Initializing the Student subclass.

We see a very familiar initializer method here, with **first_name** and **last_name** parameters. We can re-use our initializer method for these parameters by calling our parent (or "super") class's initializer. That happens on line 133. We use super to get a reference to our parent class, then we can access it's __init__() method to initialize **first_name** and **last_name**.

All we have left to do is pass the **course_name** argument to our **self.course_name** setter which is done on line 134.

We then set up a setter and getter for **course_name** in the usual way:

```
136     @property
137     def course_name(self) -> str:
138         """
139         Gets the student's registered course name
140 
141         Returns:
142             A string containing the course name the student is registered for
143         """
144         return self.__course_name  # Returns the private attribute
145 
146     @course_name.setter
147     def course_name(self, value: str):
148         """
149         Sets the student's registered course name
150         """
151         # Validate incoming data
152         try:
153             if len(value) > 25:
154                 raise ValueError(f">>> Course name must not exceed 25 characters.\n")
155             else:
156                 self.__course_name = value.title()  # Store private attribute
157         except ValueError as e:
158             IO.output_error_messages(e)
```

Figure 10: Setter and getter for the course_name property.

Nothing new here, except we use a slightly different validation approach (line 153) to test if a value is good or bad before allowing the value to be set in the property. In this case we want the course name to be less than 25 characters long.

FileProcessor

Our FileProcessor class is substantially similar to last week's assignment, so we'll just go over what's new.

read_data_from_file()

Here's the bit of code that's new this week:

```
187         # Validate file data to see if it contains the dictionary keys we expect.
188         for i, record in enumerate(json_data, start=1):  # Loop through all records in JSON data
189             for key in KEYS:  # Loop through all keys we expect to find
190                 if not key in record:  # If key doesn't exist, throw error
191                     raise ValueError(
192                         f'>>> Missing dictionary key "{key}" in record {i}. Please check {file_name} for errors.')
193 
194             # Create a new Student object and add it to the list
195             student_data.append(
196                 Student(record["FirstName"], record["LastName"], record["CourseName"]))
197 
198         return student_data  # Send the list of Student objects back to the statement that called us
```

Figure 11: What's new in read_data_from_file().

As we loop through every record in the JSON data (line 188), we do some data validation. If the data passes validation we create a new object of type Student and append that object it to the **student_data** list (line 195).

The big difference this week is that instead of making a **list** of **dict** objects, we make a list of **Student** objects.

write_data_to_file()

Here's the bit of code that's new this week:

```
229         file: IO = None
230         registrant: Student = None  # For iterating through the student_data list
231         json_data: list = []  # For holding JSON compatible data
232 
233         # Loop through the Students in student_data and convert to JSON
234         for registrant in student_data:
235             record = {"FirstName": registrant.first_name, "LastName": registrant.last_name,
236                       "CourseName": registrant.course_name}  # Format the data as a dict
237             json_data.append(record)  # Append the dict to json_data
```

Figure 12: What's new in write_data_to_file()

The big difference here is that our list of **student_data** is a list of **Student** objects, not a list of dictionaries. We can't pass that list of objects off to the json.dump method because it doesn't understand our custom **Student** objects. So we need to prep some JSON-friendly data. We do that in lines 234-237. We iterate through all the **Student** objects in our **student_data** and for each one we set the value of **record** to be a dictionary containing the **FirstName**, **LastName**, and **CourseName** properties of the Student object.

We then append that record to our **json_data** list. After iterating through all the Students, we end up with a list of dictionaries in **json_data**, which is something we can hand off to the JSON library for writing to the file like we did last week.

IO

New in the IO class this week are a couple of methods for making the output more informational. We introduce methods for printing "info" and "warnings" to the console. Info messages are printed in green and warning messages are printed in red.

```python
    @staticmethod
    def print_info(message: str) -> None:
        """
        Prints an informational message to the console in green.

        :param message: The message to be printed
        """
        print(f"\033[0;32;49m{message}\033[39m")

    6 usages
    @staticmethod
    def print_warning(message: str, newline: bool = True) -> None:
        """
        Prints a warning message to the console in red.

        :param message: The warning to be printed
        :param newline: Boolean indicating whether to add a new line at the end of the message
        """
        if newline:
            print(f"\033[0;31;49m{message}\033[39m")
        else:
            print(f"\033[0;31;49m{message}\033[39m", end="")
```

Figure 13: Methods for printing info and warning messages in colored text.

We're using ANSI codes to set text color here. This is beyond the scope of this week's assignment, but does make it easier for the user to distinguish errors and warning from informational messages.

output_student_courses()

```
309     @staticmethod
310     def output_student_courses(student_data: list) -> None:
311         """
312         Prints out the student registration data in human-readable format.
313 
314         :param student_data: list from which student data will be presented
315         """
316         IO.print_info(">>> The current data is:\n")
317         IO.print_info("First Name          Last Name           Course Name         ")
318         IO.print_info("------------------------------------------------------------------")
319         for registrant in student_data:
320             # Print each row of the table inside fixed width columns
321             print(f"{registrant.first_name[:20]:<20}{registrant.last_name[:20]:<20}{registrant.course_name[:25]:<25}")
322         IO.print_info("------------------------------------------------------------------")
```

Figure 14: What's new in output_student_courses.

We have to make a small change to our display code here since we have a list of **Student** objects this week. Lines 319-321 loops through our list of Students and prints them in fixed-width columns. You can see on line 321 we are getting the values of the properties **first_name**, **last_name**, and **course_name** of the **Student** object by accessing the getter methods that we created before.

input_student_data()

```
335             # Input user data for new student registration
336             IO.print_info(">>> Register a student for a course\n")
337             registrant = Student()  # Creates a new Student object named registrant
338             while not registrant.first_name:  # Keep trying until we get a validated input
339                 registrant.first_name = input("Enter student's first name: ")
340 
341             while not registrant.last_name:  # Keep trying until we get a validated input
342                 registrant.last_name = input("Enter student's last name: ")
343 
344             while not registrant.course_name:  # Keep trying until we get a validated input
345                 registrant.course_name = input("Enter the course name: ")
```

Figure 15: What's new in input_student_data.

This week we're storing our student data in a **Student** object. Line 337 creates an object of type **Student** and the variable **registrant** points to that object.

We then ask the user to provide a first name, and send that off to our **Student** object's setter method, which will validate the data. If the data comes back as valid, we move on to last name and so on.

If the data doesn't come back as valid, we ask again until we get a valid answer. You can see this test happening on line 338. This line sets up **while** loop that repeats forever until the value of **registrant.first_name** returns from the getter with something valid inside it.

Main logic

Our main logic didn't change this week. The functionality of the program is also the same as last week. Because of this, all the work we did this week would be considered "refactoring" our code. It behaves the same from a user's perspective, but the underlying code has been changed to either be more flexible, simplified, or otherwise improved.

Summary

In this assignment we learned:

1. How to create classes with properties
2. How to create subclasses
3. How to use inheritance
4. How to initialize properties
5. How to create property setters and getters
6. How to translate data from JSON to objects and then back to JSON
7. How to use setters and getters to access the properties of an object