Jocelyne Booth
08/14/2024
ID FDN 110 A
Assignment 07
GitHub URL: https://github.com/j-booth4/IntroToProg-Python-Mod07

Classes and Objects

Introduction

This week, we're again simplifying the functionality of our code by organizing functions into more specific classes. Additionally, we begin using inheritance to reduce repetitive code.

Defining the 2 New Classes and Functions

Classes

This week, I had to make two new classes: Person and Student. Instead of making the student_data list be a list of dictionaries or further lists, we now make student_data a list of Student objects. Using classes to create a student makes it more clear what properties and information each student stores. Also, using a Person class to be inherited by Student makes it easier to add more types of people, e.g. Teacher, in the future.

Initializer Function

The first function to make in each class was the initializer (Fig. 1). Starting with Person, each person needs a first and last name. This makes it so we can simply create a Person object by saying Person(X,Y) to create a person with first name X and last name Y. Making these properties private using the _ means we can make methods known as first_name and last_name to access these properties, but format them differently if needed.

```
# Add first_name and last_name properties to the constructor (Done)
def __init__(self, first_name:str = "", last_name:str = ""):
    '''
    This function initializes the person object.
    :param first_name: The first name property of the person.
    :param last_name: The last name property of the person.
    '''
    # Set the private variables of first_name and last_name using the parameters.
    self._first_name = first_name
    self._last_name = last_name
```

Fig. 1. The initializer function of the Person class.

The initializer of the Student object is partly inherited from the Person class by using the super() method. This lets us easily initialize the first and last name. Then, we can initialize the course similarly how we did in the Person class.

Getter Functions

The getter functions work to return specific properties of a class. For Person, we create the name of the get function to be the name of the property, e.g, the function first_name() will return the first_name property, accessed by calling the private variable. The first and last name functions are inherited by the Student class, but a course_name getter is also added.

Setter Functions

The setter function works to set the properties of an object. Within the program, once the data has been input, it uses this as a parameter to set the object property (Fig. 2). This allows us to consolidate the error handling related to entering first/last name to only contain letters. It's also a chance to modify the data before saving it, e.g. if you want to capitalize all the letters.

```
@first_name.setter
def first_name(self, name:str):
    '''
    This string sets the first name of the person.
    :param name: the string to be used for the Person's first_name property
    '''
    # sets the first name if it's alphabetical characters only.
    if name.isalpha() or name == "":
        self._first_name = name
    else:  # if numbers are present, will display an error
        raise ValueError("First name must be alphabet characters only.")
```

Fig. 2. The setter function for the Person's first_name property

The setter for the Parent are inherited by the Student once again, and I only had to create a new setter for course_name.

String Functions

Classes allow us to override the default string printed by calling an object. Using the __str__ method, I have Parent and Student return their properties as comma-separated strings. Parent and Student had separate __str__ methods since they have different properties, showing how the Student method overrides the Parent.

Modifying the FileProcessor and IO Classes

These classes originally relied on the student_data being a list of lists, but now it's a list of Student objects. So, I had to modify the read/write functions of the FileProcessor and the input/output functions of the IO.

FileProcessor

My first problem I encountered was trying to write new data into the file. I got an error since I was trying to write the Student object into the JSON-formatted file since the Student object doesn't have keys like "FirstName" or "CourseName". So, for the write function, I had to convert each student from Student object to a dictionary, then I could append it to a new list of students that could be saved to the JSON file. (Fig. 3).

```
file = open(file_name, "w")
list_of_students: list = []
# Convert the list of Students to a list of dict entries
for student in student_data:
    entry: dict = {"FirstName": student.first_name, "LastName": student.last_name, "CourseName": student.course_name}
    list_of_students.append(entry)
# write the list of JSON-formatted entries into the JSON file
json.dump(list_of_students, file)
print('Finished writing data to file. New data:')
file.close()
IO.output_student_and_course_names(student_data=student_data)
```

Fig. 3. The write_data_to_file function from the FileProcessor class.

IO

Outputting student data was the first test I tried after creating my classes. Using Menu Option 2, I immediately got an error that the student_data was of the wrong type. This is how I realized that the student_data wasn't being read as Student objects, so I had to convert the dictionary of students into Student objects. Then, I could print each student using the print(object : Student) function where the Student string now prints comma-separated data.

Inputting student data was like the reverse of the output, as I had to convert each Student object from student_data into a dictionary. This let me format it for the JSON file using keys.

Summary

By creating two new classes called Person and Student, I was able to reduce repetitive code and make it easy to create and call an object's properties. These make it very clear what attributes each student has when called within the main body of code. Using inheritance, we also make it more convenient to create similar objects but also to expand upon them.