Adam Savage

November 27, 2024

IT FDN 110 A

Assignment 07

https://github.com/adsav54/IntroToProg-Python-Mod07

Working with Data Classes

Assignment07

Introduction

In this module, we learned about classes and how to use them to organize functions. We focused on data classes and how to abstract our code to make it more robust and portable. We were introduced to constructors, properties, private attributes, and inheritance, advanced tools in Python that make code more flexible, more maintainable, and extensible.

Fundamentals of this module

Statements are code that operate on the data, test conditions, or perform loops. Functions 'wrap' groups of statements for reuse in the program and and for organization. Classes organize functions that are interrelated or of similar type for modularity across programs. Leveraging classes and functions to use statements are critical strategies as data manipulations and IO grow in complexity.

By convention, classes are grouped into three types: data, presentation, and processing. We have considered presentation and processing previously. Data classes are different from the others in that they typically have attributes, constructors, properties, and methods (ie, functions). Attributes are variables that store the data or properties of objects. Constructors are specialized methods that are automatically called when a class is invoked during object creation. Their primary purpose is to set default attributes for the object. An important feature is the use of the '*self'* keyword to assign attributes to the object being defined and not the class. Properties are functions designed to manage attribute data. A property method can be used as a pseudo instance of a variable to call the method that accesses a private variable. Private attributes are those that should not be modified outside of the class.

Classes can be chained together in Parent-Child (or Superclass-Subclass) relationship. By defining a class as a child of a parent class (***Figure 1***), default attributes and properties can be inherited.

```
class Person:...

class Student(Person):
    """A collection data about students..."""
    def __init__(self, first_name: str = '', last_name: str = '', gpa: float = 0.00):
        # self.first_name = first_name
        # self.last_name = last_name
        super().__init__(first_name=first_name, last_name=last_name)
        self.gpa = gpa
```

Figure 1: Abbreviated code showing inheritance of a parent class and its "__init__" method.

Within Python all object classes are children of the parent Object class, which is assumed and typically not explicit in the code. The Object class has default methods, such as "__init()__" and "__str()__". These default methods, and any parent method, can be used and/or overridden by child classes. By defining the same method as a parent method, the parent method is overridden. Variables from the parent can also be initialized and used in the child class by calling "super().__init__…" as in ***Figure 1***.

The Assignment07 script

In assignment 07 we converted the starter script from using dictionaries and lists of dictionaries to using object data from the Object class. The most significant change was to at data classes 'Person' and 'Student'. Each of these data classes contained an '__init__()' method initialized the parameters 'self', 'first_name', and 'last_name'. The 'Student' class was created as a child of the 'Person' class and inherited the 'Person' attributes 'first_name' and 'last_name' with the 'super()' method. 'Student' also added another attribute, 'course_name'. Both classes contained getter and setter methods for non-inherited attributes, following the rule of '@property' designation for the getter and '@[getter_name]' for the setter (***Figure 2***).

```
@property  # (Use this decorator for the getter or accessor)
def first_name(self):
    return self.__first_name.title()  # formatting code

@first_name.setter
def first_name(self, value: str):
    if value.isalpha() or value == "":  # is character or empty string
        self.__first_name = value
    else:
        raise ValueError("The last name should not contain numbers.")
```

Figure 2: Getter and setter methods for the attribute 'first_name'.

These two methods handle the variables directly after being called on in the code using '[Class.method]' nomenclature (***Figure 3***).

```
# Input the data
student = Student()
student.first_name = input("What is the student's first name? ")
```

```
student.last_name = input("What is the student's last name? ")
student.course_name = str(input("In what course is the student enrolled? "))
```
Figure 3: The code for passing user input to setter methods.

Each instance of the ‘Student’ object is then appended to the ‘students’ list, for concise data handling.

Upon establishing the use of data class objects, file processing and data output had to be modified to read the data from objects. File input and output became more complex, as JSON formatted file data was read into a list of dictionaries with ‘json.load()’ and then had to be parsed into instances of the ‘Student’ data object. On file output the reverse had to occur, with data objects converted to a list of dictionaries temporarily so the data could be written in JSON format by ‘json.dump()’. ‘print()’ statements also had to be modified to display data from data objects rather than from a dictionary (***Figure 4***).

```
for student in student_data:
    # print(f'Student {student["FirstName"]} {student["LastName"]} is enrolled in
    # {student["CourseName"]}')
    print(f'Student', student.first_name, student.last_name, 'is enrolled in', student.course_name)
```
Figure 4: The ‘print()’ statement modified to display object data (old code commented).

Summary

Here we learned about and practiced the use of the data class, with constructors, properties, private attributes, and inheritance, modifying older code that relied on lists of dictionaries. Though numerous changes were needed, the new code is cleaner, more robust and more extensible. With the use data classes we will be able to cleanly manage much more complex data structures with real-world applications.