Matthew Young

Introduction to Programming: Python

03/12/2025

GitHub URL:

Abstraction and Encapsulation

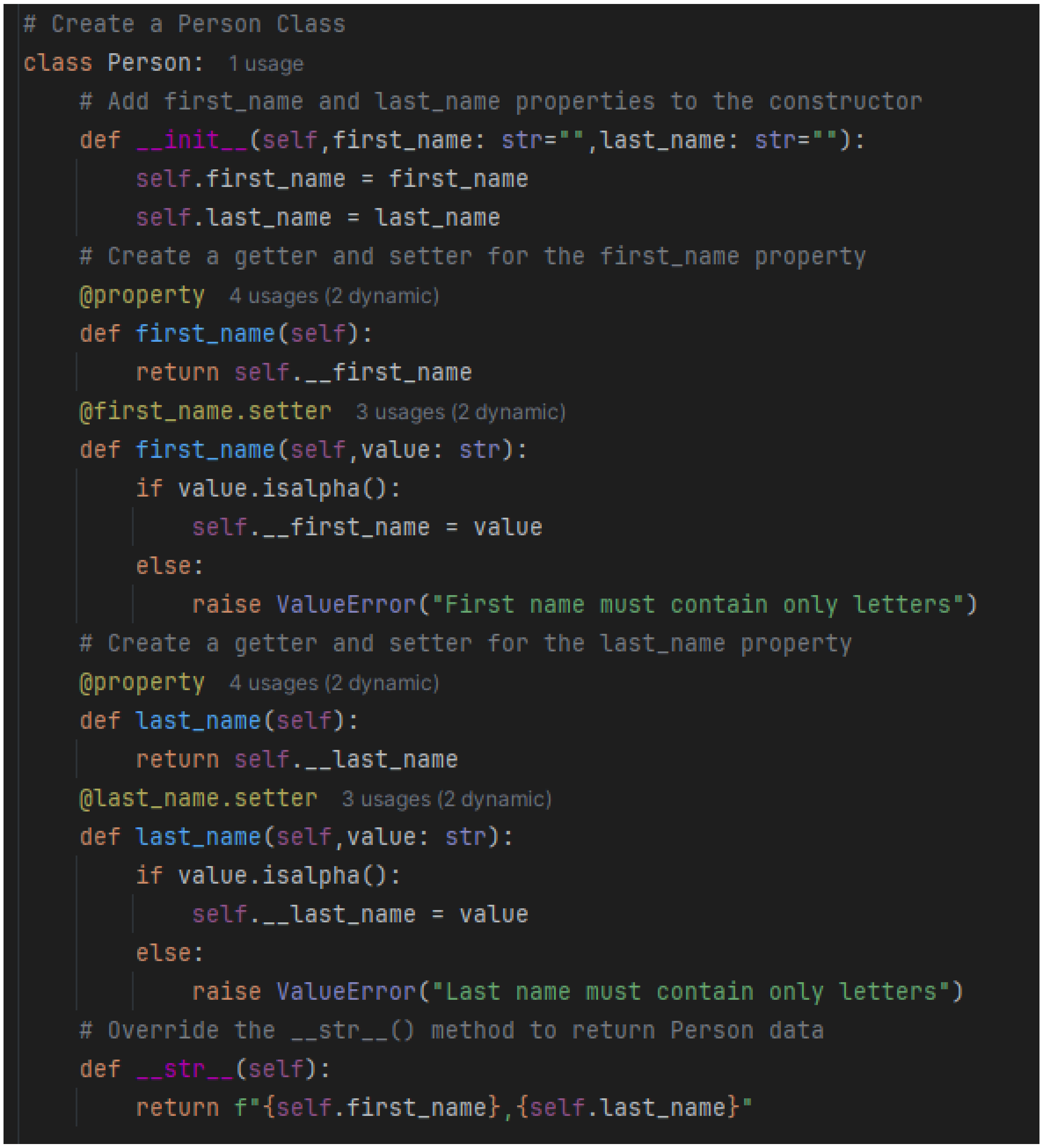
Assignment 07

Introduction

This assignment focused on data classes and the components that comprise them in Python programming to further optimize the script that's been built up over the duration of this course. Complex programs consist of a number of statements, functions, and classes and last week's assignment looked at functions and classes and the modularity they bring with their implementation in programs. This week further explored the capabilities of classes, data classes in particular, and the utilities of the features they bring, attributes and constructors for example, to produce a more robust code.

Data Classes

As last week's assignment covered, complex programs are composed of a variety of statements, functions, and classes that organize the flow and readability of a program's code. This assignment delved further into the applicability of these features. Classes organize functions as functions organize statements. They allow the developer to organize the various functions into their corresponding classes based on their corresponding purposes. Read and write to file functions for example, are separated into their own FileProcessor class in our code. The functions, or Methods, that fall under a class can all contain variables to store information. Objects are unique instances of specified classes, similar to cookie cutters, where they can be called and used to store data unique to each instance they are used. An example of this can be seen in Figure 1 with the newly implemented Person class.



```
# Create a Person Class
class Person:  1 usage
    # Add first_name and last_name properties to the constructor
    def __init__(self,first_name: str="",last_name: str=""):
        self.first_name = first_name
        self.last_name = last_name
    # Create a getter and setter for the first_name property
    @property  4 usages (2 dynamic)
    def first_name(self):
        return self.__first_name
    @first_name.setter  3 usages (2 dynamic)
    def first_name(self,value: str):
        if value.isalpha():
            self.__first_name = value
        else:
            raise ValueError("First name must contain only letters")
    # Create a getter and setter for the last_name property
    @property  4 usages (2 dynamic)
    def last_name(self):
        return self.__last_name
    @last_name.setter  3 usages (2 dynamic)
    def last_name(self,value: str):
        if value.isalpha():
            self.__last_name = value
        else:
            raise ValueError("Last name must contain only letters")
    # Override the __str__() method to return Person data
    def __str__(self):
        return f"{self.first_name},{self.last_name}"
```

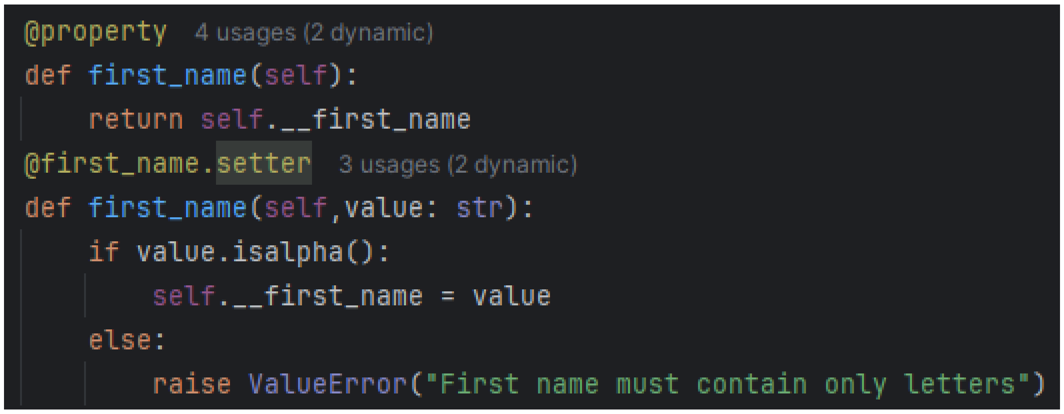
Figure 1: Defined functions to read and write data to file.

Figure 1 showcases the Person class which contains all of the data class components used to collect and store a student's first and last name. This figure also contains a lot of new components and concepts covered in this week's material. On the first line denoted with "__init__" is the constructor that initializes the local variables within this class. A constructor is a method that is automatically called by an object when it is called. This constructor initializes all the attributes of the associated class and defines a default state for when called as is. Attributes are characteristics associated with the object which describe and store information about the object they belong to. In this case, the attributes associated to the Person class are the first_name and last_name attributes, or Person.first_name and Person.last_name when called.

Each instance that these attributes are called store information that's unique to that instance without changing the local variables contained within the source code. This is called Encapsulation when each object stores data in its own memory location, which enforces isolation, reusability, and abstraction of the code.

Abstraction

Another concept that was covered in the previous figure was the decorators that precluded the first_name and last_name attributes of the Person class. These are the @property and @.setter functions which are functions designed to manage attribute data, shown in Figure 2. As the names imply, "getter" functions are how the program "gets" data from the class attribute, denoted by the @property decorator, which enables data access with optional formatting. Following this, the "setter" decorator enables data validation and error handling when obtaining data from the program. This enforces the practice called Abstraction, which is the simplification of complex algorithms by creating classes based on essential properties and behaviors that objects should have and defining what an object does rather than how it does it.

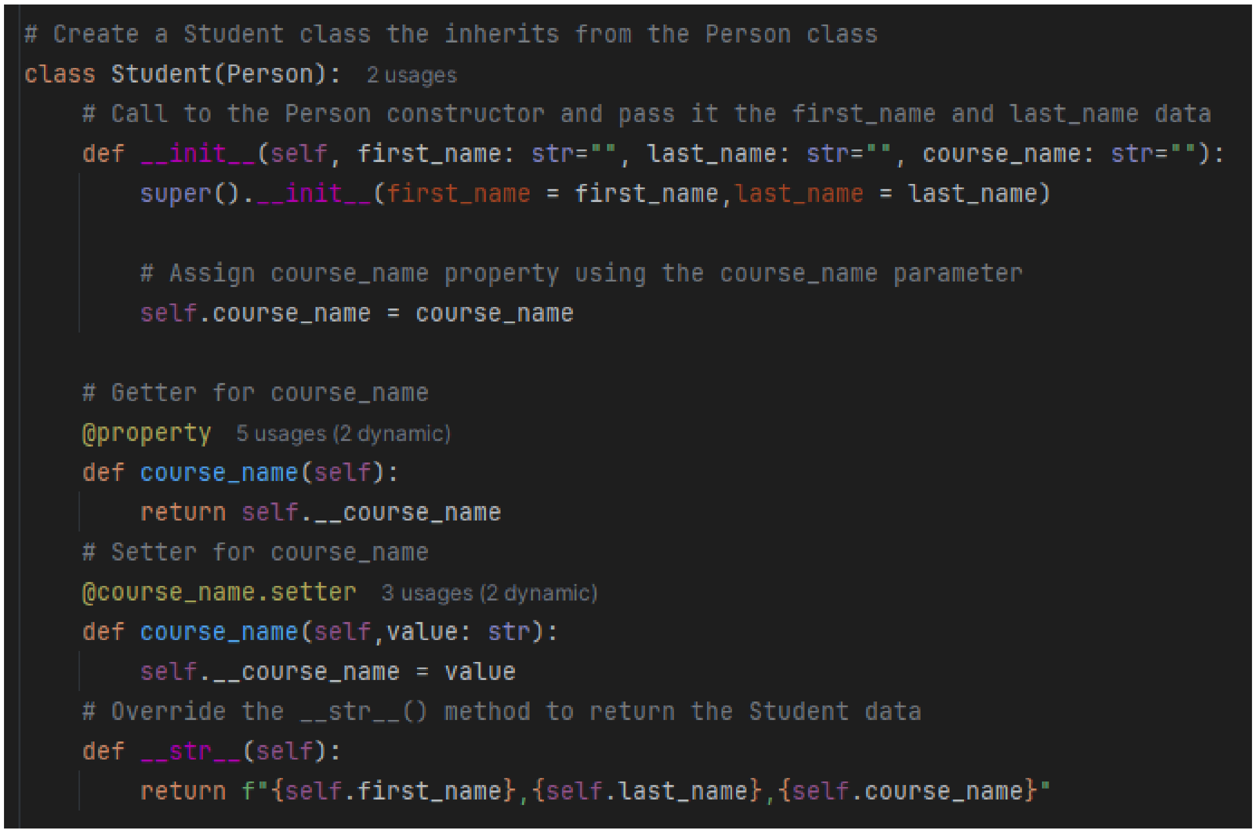


```
@property  4 usages (2 dynamic)
def first_name(self):
    return self.__first_name
@first_name.setter  3 usages (2 dynamic)
def first_name(self,value: str):
    if value.isalpha():
        self.__first_name = value
    else:
        raise ValueError("First name must contain only letters")
```

Figure 2: Data Layer of the script

Inherited Code

The final concept that was incorporated into this week's program was inherited code which allows new classes to inherit data and behavior from existing classes, establishing a class hierarchy. This can be seen in Figure 3 below where the Student class is inheriting the first_name and last_name attributes from the previously mentioned Person class. This class contains the first_name and last_name parameters like the Person class, in addition to the course_name parameter. Since the Student class is likely to use the same data for the first_name and last_name parameters as the Person class, it was easier to inherit the parameters directly from the Person class with the super().__init__ function to call the constructor of the parent Person class to initialize the parameters instead of defining them again. This concept is called Overriding Methods which allows managing changes between classes that follow similar data validation logic and error handling procedures, like catching numbers where proper names are supposed to be input.



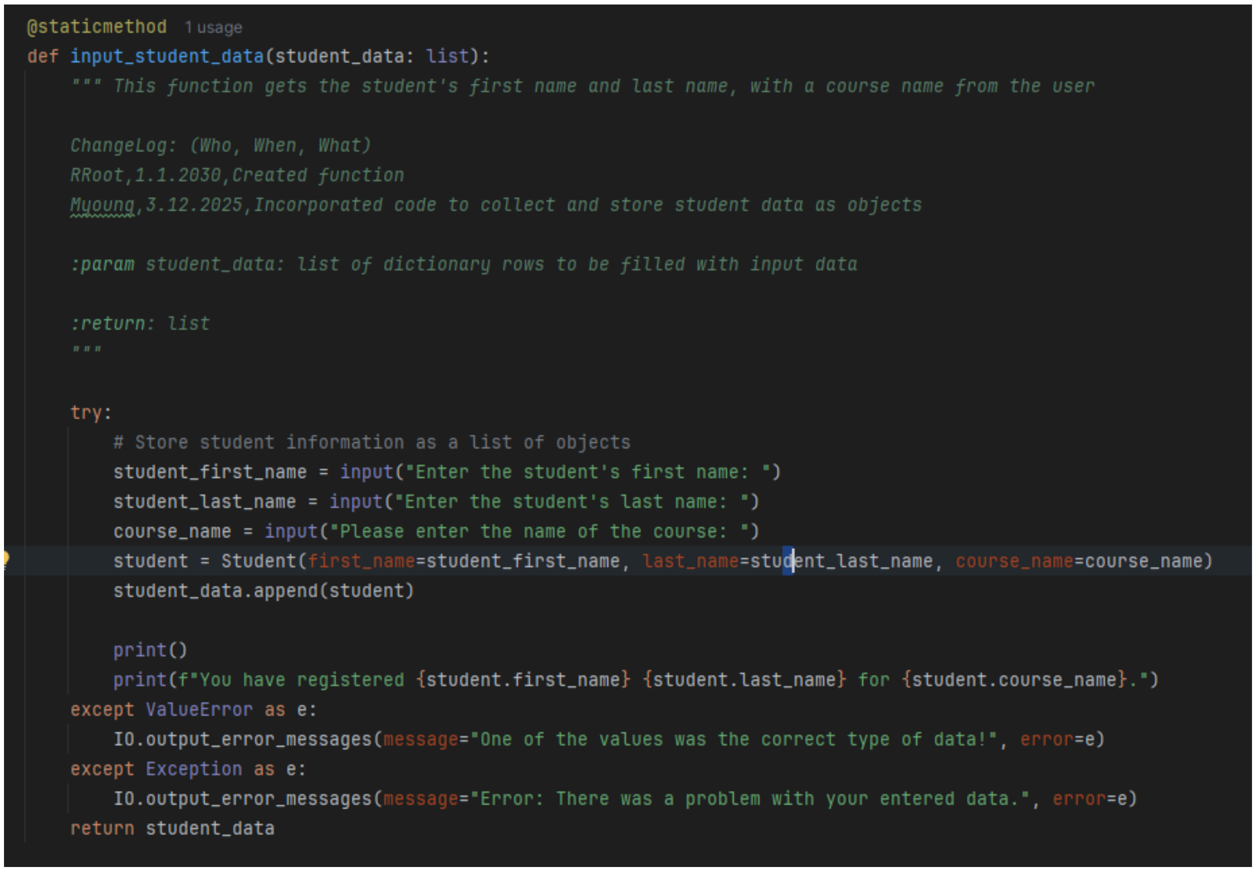
```python
# Create a Student class the inherits from the Person class
class Student(Person):  2 usages
    # Call to the Person constructor and pass it the first_name and last_name data
    def __init__(self, first_name: str="", last_name: str="", course_name: str=""):
        super().__init__(first_name = first_name,last_name = last_name)

        # Assign course_name property using the course_name parameter
        self.course_name = course_name

    # Getter for course_name
    @property  5 usages (2 dynamic)
    def course_name(self):
        return self.__course_name
    # Setter for course_name
    @course_name.setter  3 usages (2 dynamic)
    def course_name(self,value: str):
        self.__course_name = value
    # Override the __str__() method to return the Student data
    def __str__(self):
        return f"{self.first_name},{self.last_name},{self.course_name}"
```

Figure 3: Excerpt of the Presentation Layer of the code

Figure 3 also showcases the built-in __str__ method which allows the programmer to also customize the return string produced by the method when called in the class. In this case, the student’s first name, last name, and course name are all configured as a comma separated string for data storage purposes. Putting it all together in Figure 4, the input_student_data function was updated to include the newly created Student() function with customized first_name, last_name, and course_name attributes using the collected data from the previous input() functions as custom objects to the student_data list to be manipulated and formatted for future data processing.



```
@staticmethod  1 usage
def input_student_data(student_data: list):
    """ This function gets the student's first name and last name, with a course name from the user

    ChangeLog: (Who, When, What)
    RRoot,1.1.2030,Created function
    Myoung,3.12.2025,Incorporated code to collect and store student data as objects

    :param student_data: list of dictionary rows to be filled with input data

    :return: list
    """

    try:
        # Store student information as a list of objects
        student_first_name = input("Enter the student's first name: ")
        student_last_name = input("Enter the student's last name: ")
        course_name = input("Please enter the name of the course: ")
        student = Student(first_name=student_first_name, last_name=student_last_name, course_name=course_name)
        student_data.append(student)

        print()
        print(f"You have registered {student.first_name} {student.last_name} for {student.course_name}.")
    except ValueError as e:
        IO.output_error_messages(message="One of the values was the correct type of data!", error=e)
    except Exception as e:
        IO.output_error_messages(message="Error: There was a problem with your entered data.", error=e)
    return student_data
```

Figure 5: The Processing Layer of the script that comprises the main body of the script.

Conclusion

This assignment focused on further reinforcing the best practices of encapsulation and abstraction of code to improve the maintainability and organization of the program. Several new features were incorporated into the code through expanded data classes that include custom attributes and constructors to create and store data in custom objects and user-friendly return values and strings with error handling.