

Mia Bolding

08/10/2024

Introduction to Programming with Python

Assignment 07 – Programming Basics

[GitHub Link](#)

Python Classes and Objects

Introduction

This week I've learned that using parent and child classes, like Person and Student, makes programming more streamlined by organizing data and functionality more effectively. By inheriting and extending features, I can reduce redundancy and simplify my code. This approach helps in managing complex tasks more efficiently and keeps my codebase clean.

Common Definition

Statements: are instructions that Python follows to do something. They can be any line of code, like assigning a value to a variable, controlling the program's flow with loops or conditions, or running a function. For example, `print("Hello World")` is a statement, and so is `x = 3`.

Functions: are pieces of code that do a specific job. You can give them inputs, called arguments, and they can give back an output. They make your code easier to manage and save time by letting you use the same code in different places without having to rewrite it.

For example:

```
student_name = "Vic Vy"
courses_name = ["Python 200"]
register_student(student_name, courses_namer)
```

This way, if I need to register another student, I don't have to write the code again. I can just call the function with different details, making my code simpler and easier to manage.

Classes: are blueprints for creating objects, also known as instances. They can hold data (called attributes) and define behaviors (called methods or functions) related to that data. I can use classes to build complex data structures and models, organizing and grouping related code together.

For example:

```
class Student:
    def __init__(self, first_name, last_name):
        # Initialize the student's first name, last name, and courses
        self.first_name = first_name
```

```
self.last_name = last_name
self.courses = []
```

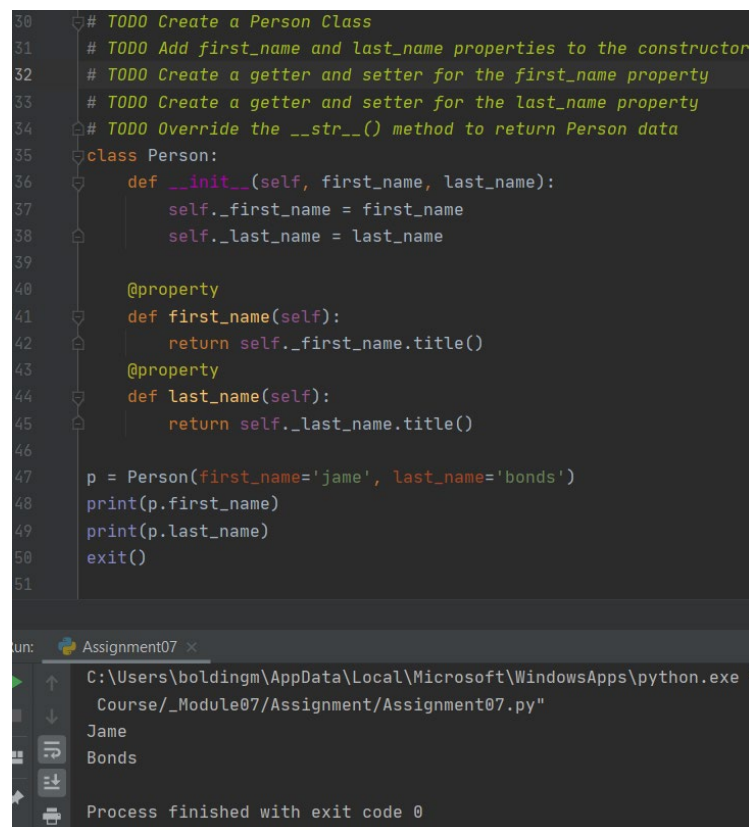
The class `Student` acts as a blueprint for creating student objects, meaning it defines what attributes and behaviors each student will have. The `__init__` method is a special setup function that automatically runs when you create a new student. It initializes the student's attributes with the values you provide. The `self` keyword is used within this method to refer to the specific student being set up, allowing me to set or access that student's details like their first name, last name, and list of courses.

Lesson Learn During Assignment

I used the assignment 07 starter which already have the `FileProcessor` and `IO` class so my job is to add `Person` and `Student` class. In this case the `Person` class is larger and will cover the `Student` class so it is the parent class while `Person` class is the smaller class that extended from the `Person` class.

Person class:

Figure 1: Write the `Person` class with getter methods that capitalize the first letter of both the first and last names. I also want to run the code to ensure it works correctly.



```
30 # TODO Create a Person Class
31 # TODO Add first_name and last_name properties to the constructor
32 # TODO Create a getter and setter for the first_name property
33 # TODO Create a getter and setter for the last_name property
34 # TODO Override the __str__() method to return Person data
35 class Person:
36     def __init__(self, first_name, last_name):
37         self._first_name = first_name
38         self._last_name = last_name
39
40     @property
41     def first_name(self):
42         return self._first_name.title()
43     @property
44     def last_name(self):
45         return self._last_name.title()
46
47 p = Person(first_name='jame', last_name='bonds')
48 print(p.first_name)
49 print(p.last_name)
50 exit()
51
```

Run: Assignment07

C:\Users\boildingm\AppData\Local\Microsoft\WindowsApps\python.exe
Course/_Module07/Assignment/Assignment07.py"

Jame
Bonds

Process finished with exit code 0

I learn that in Python, a setter method allows me to control how a property's value is set. It is part of the property mechanism that let me define custom behavior for getting and setting value of attributes. For example, in assignment 07, In my Person class, I use setters for `first_name` and `last_name` to ensure that only alphabetic values are assigned. If I provide a non-alphabetic value, the setter raises a `ValueError`. The `@property` decorator formats and capitalizes the names. I then create a Person object, set valid names, and print them.

Figure 2. Setter methods in Assignment 7 ensure that only alphabetic characters are allowed in the `first_name` and `last_name` attributes. If non-alphabetic characters are provided, they raise a `ValueError`.

```
@first_name.setter
def first_name(self, value: str):
    if value.isalpha():
        self._first_name=value
    else:
        raise ValueError("First name must be alphabetic")

@last_name.setter
def last_name(self, value: str):
    if value.isalpha():
        self._last_name=value
    else:
        raise ValueError("last name must be alphabetic")
```

Figure 3. Add numbers after `first_name` and `last_name` to test if the `RaiseError` work.

```
p = Person(first_name='jame1', last_name='bonds1')
print(p.first_name)
print(p.last_name)
exit()
```

Assignment07 ×

```
File "C:\Users\boldingm\OneDrive - Sound Transit\Mia\Le
self.first_name = first_name
^^^^^^^^^^^^^^^^^^

File "C:\Users\boldingm\OneDrive - Sound Transit\Mia\Le
raise ValueError("First name must be alphabetic")
ValueError: First name must be alphabetic
```

Student class: I set up the Student class by building on top of a Person class, and I added a course name to it. First, I make sure the student's first and last names are set using the Person class. Then, I add a new detail: the course the student is taking. So, with this code, I can create student objects that include both their name and the class they're enrolled in.

Figure 4. Student class

```
# TODO Create a Student class the inherits from the Person class (Done)
class Student(Person):
    def __init__(self, first_name:str, last_name:str, course_name):
        #overload, add course name
        super().__init__(first_name, last_name)
        self._course_name=course_name
```

Figure 5. The __str__ method of the parent class (Person).

```
@course_name.setter
def course_name(self,value)->None:
    # Set the course_name attribute with the provided value
    self._course_name=value
def __str__(self)->str:
    # Return a formatted string combining Person's __str__ output and course_name
    return f'{super().__str__()}, {self._course_name}'
```

Similar to Assignment 6, the IO class manages user interactions, including displaying menus, handling input, and showing errors. Meanwhile, the FileProcessor class handles file reading and writing. The Person and Student classes simplify data display and input operations for the IO class and help structure data that FileProcessor can easily convert to and from JSON, making file operations more straightforward.

Testing

After integrating the Person, Student, FileProcessor, and IO classes, I ran the program and tested each option from 1 to 4 to identify errors and make corrections based on suggestions. Below are the test results, including visual output and the JSON file.

```
Enter your menu choice number: 1
Enter the student's first name: Hellen
Enter the student's last name: Hao
Please enter the name of the course: Python 200

You have registered Hellen Hao for Python 200.

Enter your menu choice number: 2
-----
Student Henry Ford is enrolled in Python 100
Student Jonhanson Deers is enrolled in Python 201
Student Hellen Hao is enrolled in Python 200
-----

Enter your menu choice number: 3
-----
Student Henry Ford is enrolled in Python 100
Student Jonhanson Deers is enrolled in Python 201
Student Hellen Hao is enrolled in Python 200
-----
```

```
[
  {
    "first_name": "Henry",
    "last_name": "Ford",
    "course_name": "Python 100"
  },
  {
    "first_name": "Jonhanson",
    "last_name": "Deers",
    "course_name": "Python 201"
  },
  {
    "first_name": "Hellen",
    "last_name": "Hao",
    "course_name": "Python 200"
  }
]
```

Summary

If I had to choose one key takeaway from Module 7, it would be the `__str__` method. This method stands out to me because it allows for versatile code by reusing the string representation logic from the parent class (Person). Additionally, it ensures consistency by keeping the Student class's string representation aligned with the Person class. Lastly, it simplifies maintenance since any changes to how names are formatted need only be made in the parent class, automatically updating all subclasses without redundant code changes.