

Objects, Classes, Properties & Inheritance

GitHub repository link: <https://github.com/ST-04261/IntroToProg-Python-Module07>

Introduction

Assignment 7 expands our use of classes to include constructors, property functions, and inheritance. Last week the class was an object that stashed related variables but now it is a more active object. We introduce the special initialization function as well as properties such as getters and setters. Inheritance will allow us to start with a general class and create more specific classes without code duplication.

Objects from Classes

Last week, we had the classes `FileProcessor` and `IO`, which helped us organize our program into focused areas of concern (specifically, separating user input/output from file manipulation). We could then populate the classes with functions that dealt with specific tasks, including reading files, writing them, and processing the data into appropriate formats (lists, dictionaries, etc). This was a first step into making our code modular and reusable, but we still had a lot of global and local variables running around to keep track of.

The solution is an object instance, a type of class that will store and access data specific to some abstraction - in our case, we have `People` and `Students`. A `Student` object is generated using our `Student` class. To turn a class into an object, we introduce structures, properties, and learn more about methods beyond the `@staticmethod` that was included without comment in functions last week. Methods are the functions that operate on the data and can be called on from outside the class.

Our most generic object is the `Person`, and it included the first and last name. An object is initialized with the special function, `__init__` and introduces the `self` keyword. There can be multiple `People`, with each unique instance a single `Person`. `__init__` generates the `Person` on command, and `self` gives us a way to have multiple instances of the same variables (first name, last name) without having to initialize a new uniquely named variable for each instance.

```
# Person class is for personal data w/o course or gpa
class Person:
    def __init__(self, first_name, last_name):
        self.__first_name = first_name
        self.__last_name = last_name
```

Figure 1. The double underscore makes this instance of `__first_name` private to the class and can only be modified within the class, and is different from the `first_name` argument that was passed to the function. The user only needs to worry about providing a value for `first_name` to pass.

Constructors, Properties and Methods

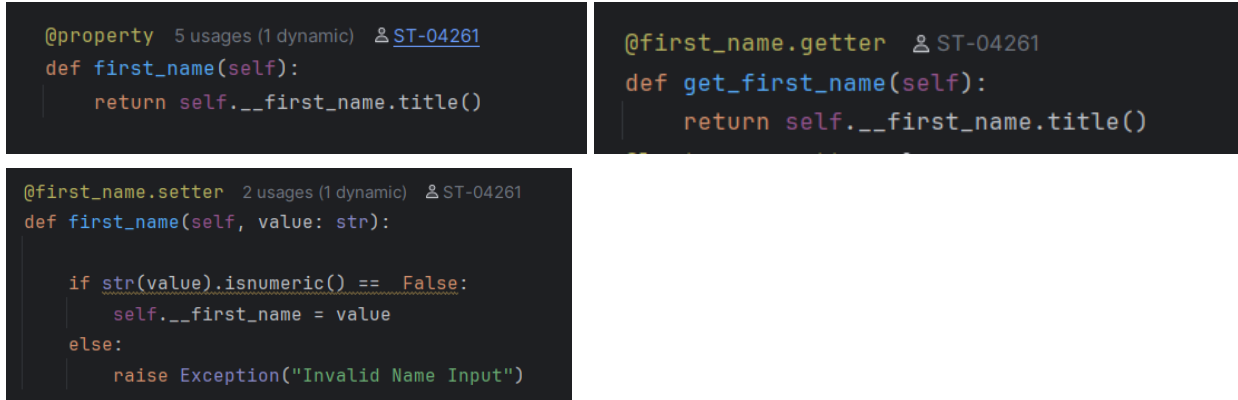
With the `__init__` function we can make many People, but they don't do anything yet. People have characteristics, and objects need properties to describe them and methods so they can perform actions.

Variables are now attributes - you are selecting and including variables that are important to the definition of the class. In this case our attributes are the first and last name. We could have more attributes that are shared by all people - age, eye color, etc - but they're not required today.

The `__init__` function is the constructor, function that builds an instance of the object when called upon.

Properties are functions that manage attributes. If we want to retrieve or change the name from a specific instance, we'll now need to go through these property functions. If you imagine back in Assignment 5 or before, we'd have to manage multiple People with a list of people, and to change the name we'd have to find the person, locate them in the list, find the proper index where that person's attribute was stored, and change the variable directly. With Objects, we reduce that one command attached to an identifying variable, like `Person1.setfirstname("Name")`.

Our properties were get and set. We now mark every function with its method. Last week, it was `@staticmethod`, used for methods that are only used within the class. Now we have `@property` and `@x.setter` and `@x.getter`.



```

@property 5 usages (1 dynamic)  ST-04261
def first_name(self):
    return self.__first_name.title()

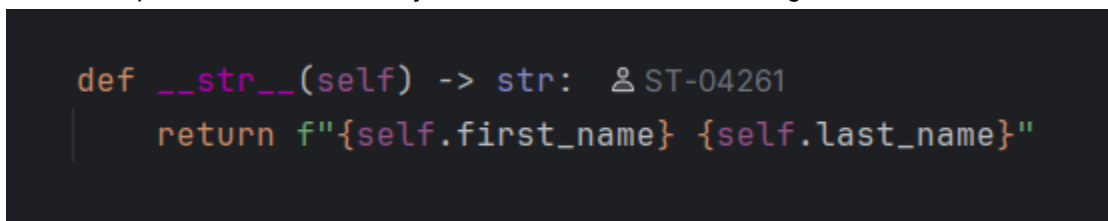
@first_name.setter 2 usages (1 dynamic)  ST-04261
def first_name(self, value: str):
    if str(value).isnumeric() == False:
        self.__first_name = value
    else:
        raise Exception("Invalid Name Input")

@first_name.getter  ST-04261
def get_first_name(self):
    return self.__first_name.title()

```

Figure 2: A property method and getter/setter pair for the first name attribute. You can see we quietly add polish like formatting the input (.title()) in the property method) so the appearance of all names will be uniform regardless of the user's capitalization choices; and error handling is included in the setter.

And finally we have what the notes call magic methods, or python-specific built-in methods that perform necessary functions for the proper operation of the class. Our two magic methods are the constructor `__init__` and our final touch, the string override, `__str__`. We need to override or replace the invisible `__str__` method that the object class bestowed upon us without mention it. Our string method needs to return the specific values of our Object in the format of our choosing.



```

def __str__(self) -> str:  ST-04261
    return f"{self.first_name} {self.last_name}"

```

Figure 3: This one messed me up in debugging because I didn't understand it completely and so just had some placeholder text in there. The part of the program that was expecting this output hated that.

Inheritance

With the people class finished, we remember we are interested in the subset of people who are students. Students have everything people have PLUS information about their courses, or GPA or academic standing or whatever (today we'll stick to courses). Fortunately, we can save a lot of effort by creating a Student class that inherits all of the attributes and functionality of the Person class, and then adds the missing attributes that are specific to this subclass.

A class that inherits from another namechecks the more generic class right at the start. It should also accept as arguments all of the attributes the parent class expects to see, in addition to its own attributes. As part of the constructor, use the super keyword to ensure you call back to the parent class as part of the initialization.

```
class Student(Person): 3 usages  ⚡ ST-04261

    def __init__(self, first_name: str='', last_name: str='', course_name: str = ''):  ⚡ ST-04261
        super().__init__(first_name = first_name, last_name = last_name)
        self.__course_name = course_name
```

Figure 4: Student inherits from Person, as you can see from the declaration. The `super()` in the initialization line reminds the class to call upon Person using the arguments the Person class expects. All other attributes are then built into the Student class just as the shared attributes were built into the Person class.

Now, everything we need to know about a student is bound in a single object instance. In this program, we have a list of these objects:

```
# TODO DONE replace this line of code to convert dictionary data to Student data
student_object: Student = Student(first_name = individual["FirstName"],
                                   last_name = individual['LastName'],
                                   course_name = individual['CourseName'])
student_objects.append(student_object)
```

Figure 4: Creating a student object from the input retrieved from a file and immediately adding that object to a list.

When we want to retrieve attributes from a specific instance of that object, that code now looks like this:

```
for individual in student_data:
    temp = {"FirstName": individual.first_name, "LastName": individual.last_name, "CourseName": individual.course_name}
    student_dictionary.append(temp)
```

Figure 5: Cycling through our updated Student Object List to create a new dictionary for our JSON file.

Summary

Objects are a critical part of writing modular, abstracted code. Inheritance allows us reuse code in multiple object types that share certain generic attributes in common. By abstracting our student list into objects, we eliminate a lot of variable tracking and juggling, with the most finicky parts of our file processing now being the parts where we have to accommodate the JSON read and write formatting.

Because we have passed attributes and private variables in addition to local/global etc, we have the benefit of tracking everything by using the same labels (`first_name`, etc) which can help us follow our path through the classes. We do need to remember that these similarly-labeled variables are not the same thing though, so changing `first_name` in the main code body does nothing to all of the `self.__first_names` or even `self.first_names` in the individual instances, at least until you have called the setter functions for each instance.

And finally, we spent a few minutes linking our PyCharm directly to GitHub so that we can upload our assignments at the push of a button.

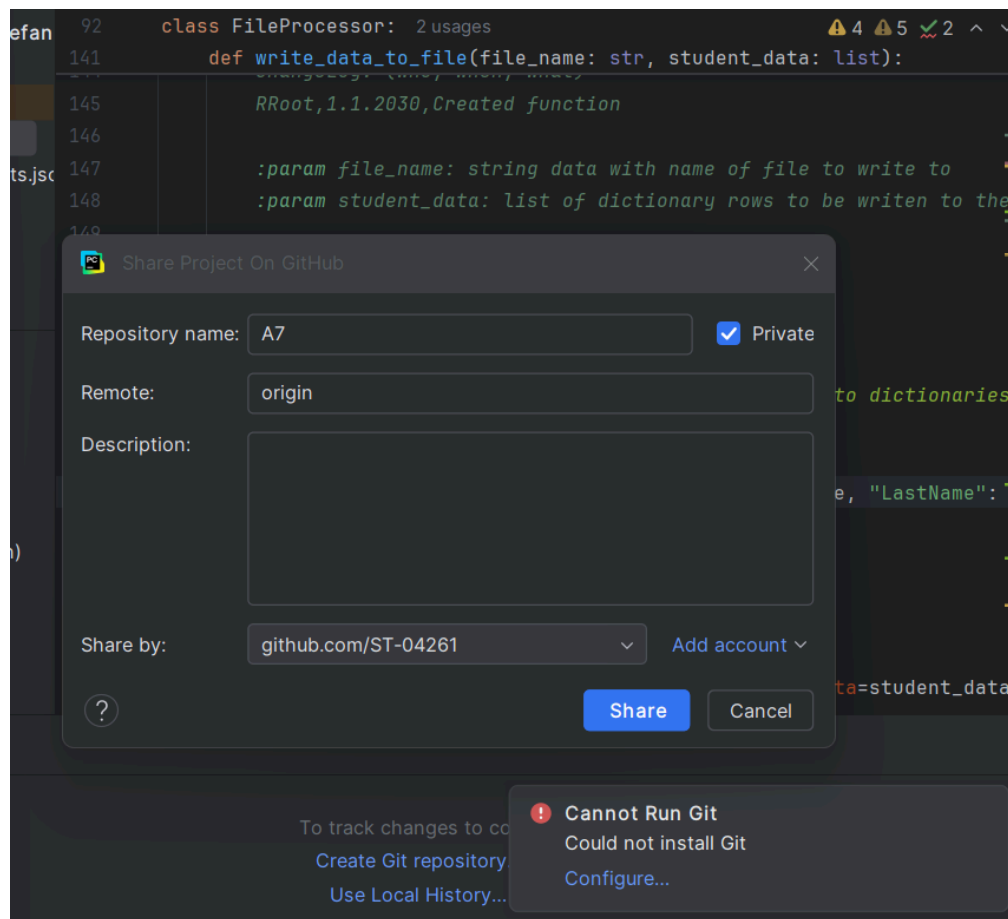


Figure 6: Ignore that error message I eventually figured it out.