Name: Wen Jiang

Date: December 9, 2025

Course: FDN 110

Assignment: Assignment 07

Assignment 07 - Classes and Objects

Introduction

In this week's course, we clarified distinctions between statements, functions, and classes. It was very helpful because in the previous modules they seemed very similar and made them easily confused for me. We also looked into classes in a greater detail, learning about the unique purpose of each class. We are also learning about object-oriented programming, which includes constructors, attributes, properties, and class inheritance.

Classes and Functions

When we want to use functions in many programs, they are organized into classes, and one program can have multiple statements, functions, and classes, depending on the complexity of the program, making programs easier to create and manage. Each object created has its own memory space and maintains its own set of data, in the code below, we have a class being defined as "Person", and the two objects created are person1 and person2, they are instances of Person "template" but represent two different people.

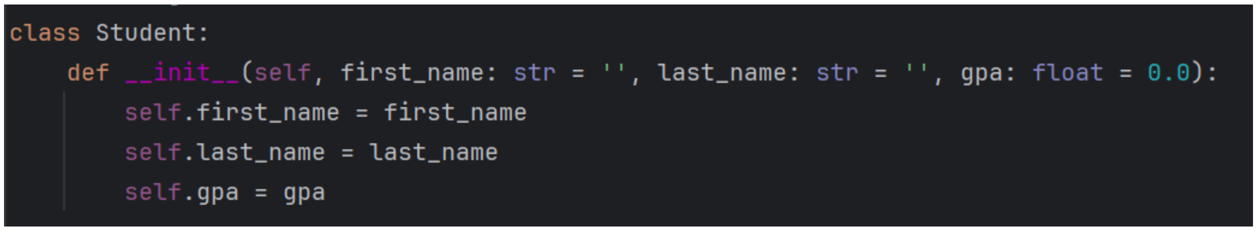
```
class Person:
    first_name: str = ''
    last_name: str = ''

person1 = Person()
person1.first_name = "Vic"
person1.last_name = "Vu"
```

Classes are focused on either data, processing, or presentation. Data classes have attributes, constructors, properties, and methods. Attributes hold data specific to objects, and have different data types, they are used to store the state of an object.

Object-Oriented Programming

In constructors, we use `def __init__(self,)` to define the class's initializer, and it will run automatically whenever we create a new object, any attributes we attach to self becomes the object's initial state.

In the figure 2.1, we can see that class is being defined as Student, given 2 objects, first_name and last_name, next, the self takes these two parameters, and makes them permanent variables stored on the object. Now we can create Student1/2/3/etc…each object will be stored in memory, with different memory addresses.



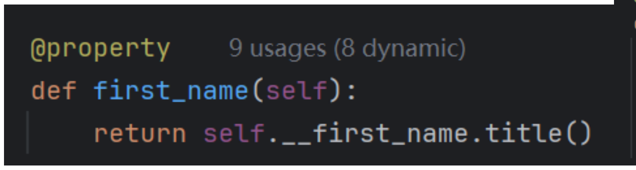
```python
class Student:
    def __init__(self, first_name: str = '', last_name: str = '', gpa: float = 0.0):
        self.first_name = first_name
        self.last_name = last_name
        self.gpa = gpa
```

Figure 1.1 self will have access to first_name and last_name, both becomes instance attribute

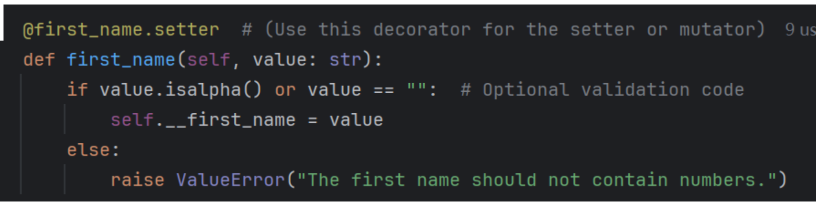
The "self" keyword is used to refer to data found in an object instance, not in class. However, we can't pass an argument to "self", and when a method is used directly, we leave out "self" and mark the method with @staticmethod.

We can also use private attributes (using self.__) if we want to hide from outside access, they can help protect the internal state of an object, preventing outside code from accidentally changing important variables, but it cannot control access by itself.

To control access to private attributes, we can use properties, they are designed to manage attribute data. 2 properties are created for each, one for "getting" data (Accessors), the other for "setting" data (Mutators).



```python
@property    9 usages (8 dynamic)
def first_name(self):
    return self.__first_name.title()
```



```python
@first_name.setter  # (Use this decorator for the setter or mutator)  9 us
def first_name(self, value: str):
    if value.isalpha() or value == "":  # Optional validation code
        self.__first_name = value
    else:
        raise ValueError("The first name should not contain numbers.")
```

Figure 1.2 & 1.3 Getter property (left) enables the access to data while optionally applying formatting. Setter property (right) allows to add validation and error handling, must include @name_of_property.setter

Properties and self.__ often work together, the private attribute hides the actual data, the property getter and setter controls the read and write access, and will only update the private value when the value passes validation.

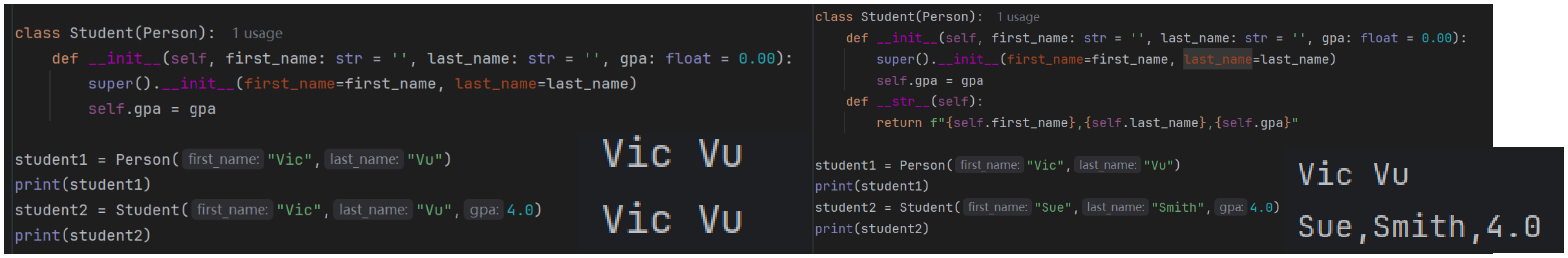
Both properties and private attributes are part of encapsulation; it focuses on how the object stores its data internally, restricting direct access to the object's components and preventing accidental modification of data. It can perform abstraction, which only shows essential features while hiding unnecessary details, giving users a clean interface, and hides how things work internally.

Inheritance

Inheritance is another OOP where a new class allows one class (child/derived class) to inherit the attributes and methods of another class (parent/base class), letting programmers to reuse code, extend functionality, and avoid duplication.

"Object" classes have several built-in methods, like int() constructor and __str__() "to string" method, and can handle core behaviors. They can be automatically invoked to respond to specific operations. These can be inherited even though we didn't create one, and we can override them to customize how the class behaves, we can change our starting values, and each data type's default values.

Overriding changes the child class to its own version, shown below.



```
class Student(Person):  1 usage
    def __init__(self, first_name: str = '', last_name: str = '', gpa: float = 0.00):
        super().__init__(first_name=first_name, last_name=last_name)
        self.gpa = gpa

student1 = Person( first_name: "Vic", last_name: "Vu")
print(student1)
student2 = Student( first_name: "Vic", last_name: "Vu", gpa: 4.0)
print(student2)
```

```
Vic Vu
Vic Vu
```

```
class Student(Person):  1 usage
    def __init__(self, first_name: str = '', last_name: str = '', gpa: float = 0.00):
        super().__init__(first_name=first_name, last_name=last_name)
        self.gpa = gpa
    def __str__(self):
        return f"{self.first_name},{self.last_name},{self.gpa}"

student1 = Person( first_name: "Vic", last_name: "Vu")
print(student1)
student2 = Student( first_name: "Sue", last_name: "Smith", gpa: 4.0)
print(student2)
```

```
Vic Vu
Sue,Smith,4.0
```

Figure 2.1 & 2.2 Left shows the inherited script, the data for gpa was ignored, it only copies what the parent class says. On the right side, after adding the code for override, we can change every data entered in this section, including adding new information.

Inheritance is important when we work on more complex programs, all objects in these classes will inherit the properties and the parent functionality, we can include additional properties depending on different needs. This makes changing easier, because the changes to the parent class will affect all child classes.

Summary

Functions organize reusable statements, and classes organize related functions and data. OOP uses encapsulation and private attributes to protect values and control access. Python classes automatically inherit built-in behaviors from constructors and magic methods like __str__(). They let objects respond to common operations by default, but can be overrode by redefining, allowing programmers to customize how objects are created and displayed.

This module also explains about Git and GitHub. I find Git is very useful, because it always saves a copy of the older files when you upload an updated one. We are able to keep track of every change we made for the project, create a copy of one file or the entire project (repository), create separate lines of development within the project with branching, and merging the branches when needed, create "commits", interact with remote repositories (push changes made by self and pull change made by others).

Below is my github link for assignment 7.