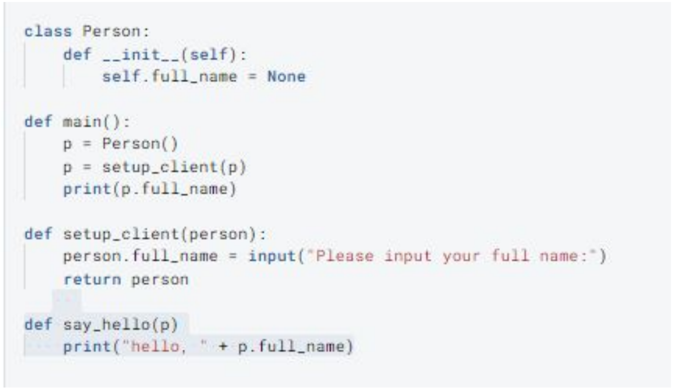
Lesson 7 - OOP

Getters and Setters

Why use getters and setters?

- Let’s say you have a specific attribute that is used in many different places outside of your class.
- For example, you have a Person class with an attribute full_name that is used throughout the program - outside of your Person class.

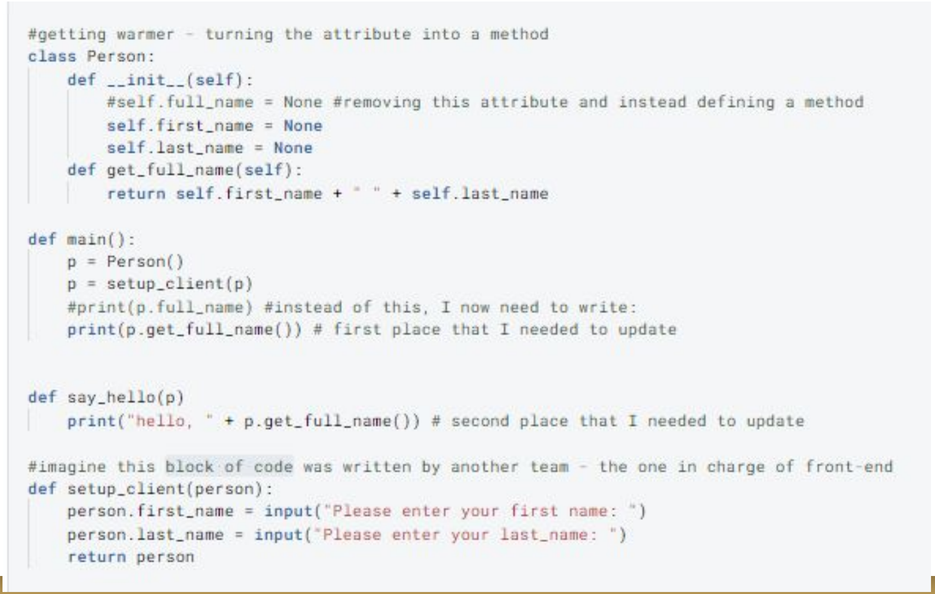


```
class Person:
    def __init__(self):
        self.full_name = None

def main():
    p = Person()
    p = setup_client(p)
    print(p.full_name)

def setup_client(person):
    person.full_name = input("Please input your full name:")
    return person

def say_hello(p)
    print("hello, " + p.full_name)
```

https://deepnote.com/workspace/Bassie%20Witkin-511f62db-d864-49b6-b1b3-17e7caebe742/project/Course-2022-85b2d840-de20-4fd8-8c72-2d6b125bac85/notebook/4%20-%20OOP%20-%20Person%20-%20getters%20and%20setters-cc824dbd2af748d7a1acf0b45b55ffbb

- Now let’s say a need arises to start storing a first_name and a last_name. You will add first_name and last_name attributes to your Person class. And what happens to full_name? Ideally, full_name should be first_name, space, last_name. This means that you are trying to turn full_name from a **stored attribute** to a **computed attribute.** How can we do this?

Why use getters and setters? (cont)

- The problem: how will it become a computed attribute if it’s just a variable?
  I could make it into a method, but It would mean I have to find every place that gets full_name and update it to call a method instead of getting an attribute.
- This will become quite cumbersome if full_name was referenced in many different places.
- 

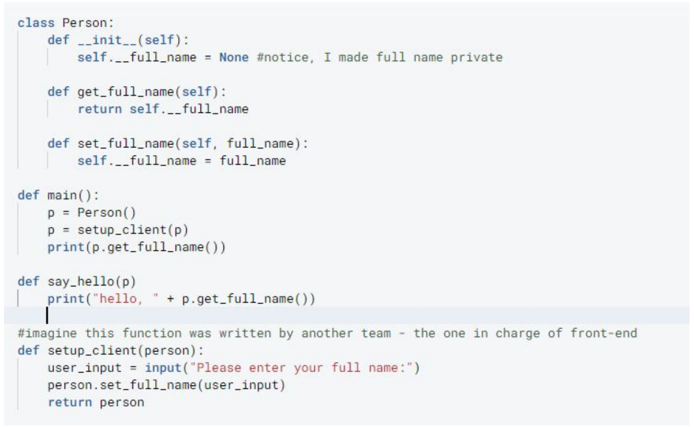
```
#getting warmer - turning the attribute into a method
class Person:
    def __init__(self):
        #self.full_name = None #removing this attribute and instead defining a method
        self.first_name = None
        self.last_name = None
    def get_full_name(self):
        return self.first_name + " " + self.last_name

def main():
    p = Person()
    p = setup_client(p)
    #print(p.full_name) #instead of this, I now need to write:
    print(p.get_full_name()) # first place that I needed to update


def say_hello(p)
    print("hello, " + p.get_full_name()) # second place that I needed to update

#imagine this block of code was written by another team - the one in charge of front-end
def setup_client(person):
    person.first_name = input("Please enter your first name: ")
    person.last_name = input("Please enter your last_name: ")
    return person
```

Why use getters and setters? (cont)

- Therefore, as a best practice, we make it a habit of using **getters** and **setters** (**accessors** and **mutators**) to retrieve and update our attributes in all cases, regardless of whether there needs to be more functionality.
- In other words, if from the beginning, we defined a method get_full_name that simply returned self.full_name, when the requirements changed, we would only have to update one line of code instead of updating everywhere the attribute is used.

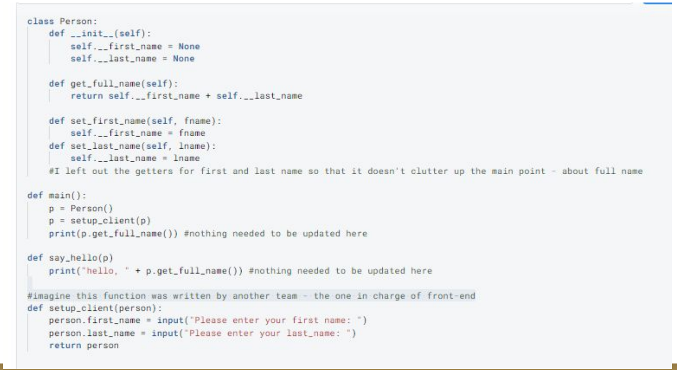
Why use getters and setters? (cont)

- To illustrate, had I originally implemented the Person class like this (before first and last name were required)



```python
class Person:
    def __init__(self):
        self.__full_name = None #notice, I made full name private

    def get_full_name(self):
        return self.__full_name

    def set_full_name(self, full_name):
        self.__full_name = full_name

def main():
    p = Person()
    p = setup_client(p)
    print(p.get_full_name())

def say_hello(p)
    print("hello, " + p.get_full_name())

#imagine this function was written by another team - the one in charge of front-end
def setup_client(person):
    user_input = input("Please enter your full name:")
    person.set_full_name(user_input)
    return person
```

- Then I would not need to update so much code with my new requirement:



```python
class Person:
    def __init__(self):
        self.__first_name = None
        self.__last_name = None

    def get_full_name(self):
        return self.__first_name + self.__last_name

    def set_first_name(self, fname):
        self.__first_name = fname
    def set_last_name(self, lname):
        self.__last_name = lname
    #I left out the getters for first and last name so that it doesn't clutter up the main point - about full name

def main():
    p = Person()
    p = setup_client(p)
    print(p.get_full_name()) #nothing needed to be updated here

def say_hello(p)
    print("hello, " + p.get_full_name()) #nothing needed to be updated here

#imagine this function was written by another team - the one in charge of front-end
def setup_client(person):
    person.first_name = input("Please enter your first name: ")
    person.last_name = input("Please enter your last_name: ")
    return person
```

Why use getters and setters? (cont)

So far, we have addressed 2 reasons to use getters and setters

1) Security - so that my attributes will be **private** to instances of my class
2) Maintainability - so that when my requirements change, I will have to update as few lines of code as possible.

From now on, only submit code whose attributes have getters and setters (unless you are sure that no one in the future will need to attach functionality)

Independent Assignment

Take ShoppingAssignment.py and alter it to use at least 2 objects.

Each object should have at least 3 instance attributes and getters and setters for every attribute (and of course an __init__() method. You can add other functionality to the object optionally.

Make sure that the output remains the same. The program output in the terminal should look identical to when there were no objects.

For more hints, click here:

For a review of today's lesson: https://realpython.com/python-getter-setter/