

# Day 36



## Getters and Setters in Python:

### What are getters and setters?

- Getter: A method used to access the value of a private attribute.
- Setter: A method used to modify the value of a private attribute.

In Python, we usually make an attribute private by prefixing it with an underscore \_ (convention) or double underscore \_\_ (name mangling).

Example: without using getter and setter. Here, nothing prevents setting age to a negative number. That's why we use getters and setters.

A screenshot of a code editor showing a Python script named main.py. The code defines a Person class with an \_\_init\_\_ method that initializes name and age attributes. It then creates a Person object p with name "Alice" and age 25, prints p.age (outputting 25), changes p.age to -5, and prints p.age again (outputting -5). Below the code editor is a terminal window showing the command 'python .\main.py' being run, followed by the output '25' and '-5'. The terminal tabs at the bottom are PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is underlined), and PORTS.

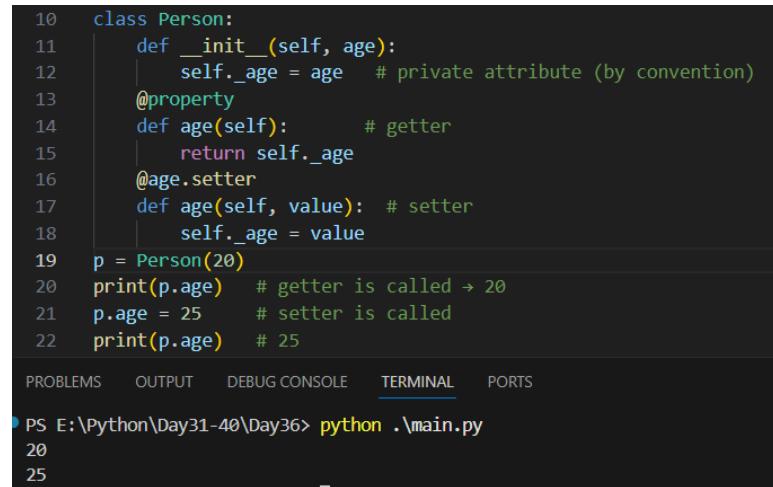
```
1  class Person:  
2      def __init__(self, name, age):  
3          self.name = name # public attribute  
4          self.age = age # public attribute  
5  p = Person("Alice", 25)  
6  print(p.age) # 25  
7  p.age = -5 # Oops! Age shouldn't be negative  
8  print(p.age) # -5
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

● PS E:\Python\Day31-40\Day36> python .\main.py  
25  
-5

This example shows a problem with public attributes. The Person class allows direct access to age, so after creating `p = Person("Alice", 25)`, you can freely change `p.age` to `-5`. Python does not stop this, even though a negative age doesn't make sense. This is why properties (getters and setters) are useful—they let you add validation and protect data while still keeping the code easy to use.

Example:



```
10  class Person:
11      def __init__(self, age):
12          self._age = age    # private attribute (by convention)
13
14      @property
15      def age(self):      # getter
16          return self._age
17
18      @age.setter
19      def age(self, value): # setter
20          self._age = value
21
22  p = Person(20)
23  print(p.age)    # getter is called → 20
24  p.age = 25      # setter is called
25  print(p.age)    # 25
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS E:\Python\Day31-40\Day36> python .\main.py  
20  
25

This code shows how properties work in Python to control access to class attributes. The Person class stores the age in a “private” variable `_age`. The `@property` decorator makes the `age()` method act like a normal attribute, so `p.age` calls the getter and returns the value. The `@age.setter` decorator lets you update the value using `p.age = 25`, which calls the setter method. This way, you can safely get and set values while still using simple attribute-style access.

Summary:

**A getter returns a value, a setter changes a value — and `@property` lets you use them like normal variables.**

--The End--