**Getters and Setters in Python**

In Python, **getters** and **setters** are methods used to access and modify the values of private attributes in a class. They help control how data is accessed and updated, ensuring encapsulation and data integrity.

**1. Why Use Getters and Setters?**

* **Encapsulation**: Hides the internal representation of an attribute and restricts direct access.
* **Validation**: Allows adding validation logic before updating an attribute.
* **Flexibility**: Makes it easy to change the internal implementation without affecting external code.

**2. Implementing Getters and Setters Using Methods**

A traditional way to implement getters and setters is by defining explicit methods.

class Person:

def \_\_init\_\_(self, name):

self.\_name = name # Private attribute (by convention, using underscore)

# Getter method

def get\_name(self):

return self.\_name

# Setter method

def set\_name(self, new\_name):

if isinstance(new\_name, str) and len(new\_name) > 0:

self.\_name = new\_name

else:

raise ValueError("Name must be a non-empty string")

# Example usage

person = Person("Alice")

print(person.get\_name()) # Output: Alice

person.set\_name("Bob")

print(person.get\_name()) # Output: Bob

**3. Using @property Decorator (Pythonic Way)**

Python provides a more elegant way to implement getters and setters using the @property decorator.

class Person:

def \_\_init\_\_(self, name):

self.\_name = name # Private attribute

@property

def name(self):

"""Getter method"""

return self.\_name

@name.setter

def name(self, new\_name):

"""Setter method"""

if isinstance(new\_name, str) and len(new\_name) > 0:

self.\_name = new\_name

else:

raise ValueError("Name must be a non-empty string")

# Example usage

person = Person("Alice")

print(person.name) # Output: Alice (Getter is called)

person.name = "Bob" # Calls the setter

print(person.name) # Output: Bob

# person.name = "" # This will raise ValueError

**4. Read-Only Property**

If you want an attribute to be **read-only** (i.e., no setter method), simply define only the getter:

class Person:

def \_\_init\_\_(self, name):

self.\_name = name

@property

def name(self):

return self.\_name # No setter method

# Example usage

person = Person("Alice")

print(person.name) # Output: Alice

# person.name = "Bob" # This will raise an AttributeError

**5. Summary**

| **Approach** | **Pros** | **Cons** |
| --- | --- | --- |
| **Direct Attribute Access** | Simple, easy to use | No encapsulation, no validation |
| **Explicit Getters and Setters** | Allows validation, encapsulation | Verbose, less Pythonic |
| **@property Decorator** | Clean, Pythonic, provides validation | None (preferred approach) |
| **Read-Only Property** | Prevents modification | Not flexible if you later need setters |

Using @property is the **most Pythonic** way to implement getters and setters in Python.