Using Properties to Control Attribute Access

October 13, 2025

Basic Property (@property)

Objective: To understand how to create a getter method for an attribute using the @property decorator, which allows a method to be accessed like an attribute.

Theory: In Python, the @property decorator is a powerful tool for creating "smart" attributes. It transforms a method into a read-only property, meaning you can access it like a variable without using parentheses. This is a common form of encapsulation, allowing you to hide internal logic from the user while still providing a simple interface. The internal, "private" attribute is typically prefixed with a single underscore (_).

Program:

```
class Student:
    def __init__(self, name, age):
        self._name = name  # A "protected" attribute
        self._age = age    # A "protected" attribute

    # The @property decorator turns this method into a getter for 'name'.
    @property
    def name(self):
        """Getter for the student's name."""
        return self._name

    # The @property decorator turns this method into a getter for 'age'.
    @property
    def age(self):
        """Getter for the student's age."""
        return self._age

# Create a Student object.
```

```
student1 = Student("Alice", 20)


# Access the properties like attributes, without using parentheses.
print(f"Student Name: {student1.name}")
print(f"Student Age: {student1.age}")
```

Output:

```
Student Name: Alice
Student Age: 20
```

Adding a Setter (@property.setter)

Objective: To learn how to add a setter method to a property to control how an attribute's value is modified.

Theory: While @property creates a read-only attribute, you can make it writable by defining a corresponding setter method. The setter is defined by using the @<property_name>.setter decorator. This allows you to add validation logic to ensure that an attribute is only set to a valid value, preventing errors and maintaining data integrity.

Program:

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, value):
        if not isinstance(value, (int, float)) or value < 0:
```

```
            raise ValueError("Width must be a non-negative number.")
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        if not isinstance(value, (int, float)) or value < 0:
            raise ValueError("Height must be a non-negative number.")
        self._height = value

# Create a Rectangle object.
rect = Rectangle(10, 5)

# Try to set valid values.
rect.width = 15
print(f"New width: {rect.width}")

# Try to set an invalid value, which will raise an error.
try:
    rect.width = -5
except ValueError as e:
    print(f"Error: {e}")

try:
    rect.height = "invalid"
except ValueError as e:
    print(f"Error: {e}")
```

Output:

```
New width: 15
Error: Width must be a non-negative number.
Error: Height must be a non-negative number.
```

Computed Properties

Objective: To use @property to create a "computed" attribute whose value is derived from other attributes and is not stored directly.

Theory: Properties are not limited to wrapping existing attributes. You can also use them to create **computed attributes**. These are attributes that are calculated on the fly when they are accessed. For example, a total_price property might calculate its value based on a price and a quantity attribute. This is a clean way to provide a dynamic value without having to manually update a variable every time a change occurs.

Program:

```
class ShoppingCart:
    def __init__(self):
        self._items = {} # Dictionary to store item and quantity

    def add_item(self, item, quantity, price):
        self._items[item] = {"quantity": quantity, "price": price}

    # A computed property that calculates the total cost.
    @property
    def total_cost(self):
        total = 0
        for item in self._items.values():
            total += item["quantity"] * item["price"]
        return total
```

```
# Create a shopping cart.
cart = ShoppingCart()
cart.add_item("Apple", 5, 0.50)
cart.add_item("Banana", 3, 0.25)

# Access the computed property.
print(f"Total cost of cart: ${cart.total_cost}")

# Add another item and see the total cost automatically update.
cart.add_item("Orange", 2, 0.75)
print(f"Total cost after adding oranges: ${cart.total_cost}")
```

Output:

```
Total cost of cart: $3.25
Total cost after adding oranges: $4.75
```