roperty decorators, which are one of Python's most elegant features for *controlled attribute access*.

Let's walk through this problem step by step.

🚗 Problem 8: Make `model` Read-Only Using `@property`

```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand      # private attribute
        self.__model = model      # private attribute

    @property
    def model(self):
        """Read-only property for model"""
        return self.__model

    def full_name(self):
        return f"{self.__brand} {self.__model}"
```

✅ Usage Example

```
my_car = Car("Tata", "Safari")

# Accessing model (works fine)
print(my_car.model)     # ✅ "Safari"

# Trying to change model (will fail)
my_car.model = "Nexon"  # ❌ Raises AttributeError
```

🧾 Output:

```
Safari
Traceback (most recent call last):
```

```
  ...
AttributeError: can't set attribute 'model'
```

🧠 Explanation

- `@property` turns a **method** into an **attribute-like getter**.
  So instead of calling `my_car.model()`, you can just write `my_car.model`.
- You didn't define a **setter** for it, which makes it **read-only**.

💡 How It Works Under the Hood

When you write:

```
my_car.model
```

Python actually calls:

```
Car.model.__get__(my_car)
```

When you try to assign:

```
my_car.model = "Nexon"
```

Python looks for a setter (`@model.setter`).
Since you didn't define one → ❌ raises `AttributeError`.

🔧 Optional (If You Later Want to Allow Setting It)

You can safely add a setter like this:

```
@model.setter
```

```
def model(self, new_model):
    print("You are changing the model!")
    self.__model = new_model
```

Now the attribute becomes *read-write* again — but you still maintain control and can add validation if needed.

🧩 In summary

Concept	Description
`@property`	Defines a getter method (read-only if no setter defined)
`@<name>.setter`	Optional — defines write logic
Benefit	Controls access to private data while keeping syntax clean

Perfect 👍 Let's now extend your understanding to the **complete property control** system in Python — using `@property`, `@<name>.setter`, and `@<name>.deleter`.

We'll build it on top of your **Car** example so it stays consistent with what you've learned.

🚗 Full Example — Getter, Setter, and Deleter

```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand
        self.__model = model
```

```
    # ✅ Getter – read-only access initially
    @property
    def model(self):
        return self.__model

    # ✅ Setter – allows controlled modification
    @model.setter
    def model(self, new_model):
        if len(new_model) < 3:
            raise ValueError("Model name too short!")
        print(f"Changing model from {self.__model} → {new_model}")
        self.__model = new_model

    # ✅ Deleter – allows controlled deletion
    @model.deleter
    def model(self):
        print(f"Deleting model {self.__model}")
        del self.__model
```

🧪 Usage Example

```
my_car = Car("Tata", "Safari")

print(my_car.model)      # ✅ Access (getter)
my_car.model = "Nexon"   # ✅ Modify (setter)
print(my_car.model)

del my_car.model         # ✅ Delete (deleter)
```

🧾 Output

```
Safari
Changing model from Safari → Nexon
Nexon
Deleting model Nexon
```

🧠 Concept Summary

Decorator	Purpose	Example
`@property`	Getter — defines how to **access** the attribute	`my_car.model`
`@model.setter`	Setter — defines how to **modify** the attribute	`my_car.model = "Nexon"`
`@model.deleter`	Deleter — defines what happens when you **delete** the attribute	`del my_car.model`

🪄 Why This Is Useful

- You can add **validation**, **logging**, or **restrictions** without changing how your class is used.
- The syntax looks **clean**, but under the hood, you have full control.

Excellent question 🔥 — this shows you're thinking **deeply** about how Python works internally.

Let's break it down clearly 👇

🧩 The doubt

> "Function name and variable name are same — so why doesn't it clash?"

You're referring to this part:

```
@property
def model(self):
    return self.__model
```

Here, both the **function name** (`model`) and the **attribute name used in code** (`my_car.model`) look the same.

So why no conflict? 🤔

🧠 The secret: Descriptor Protocol

When you write:

```
class Car:
    @property
    def model(self):
        return self.__model
```

What happens internally is:

- The decorator `@property` **wraps the function** `model()` into a **special object** of type `property`.
- So in the class dictionary, `Car.__dict__['model']` is no longer a function — it's a `property` object, not the same as the instance variable.

Let's see this in action:

```
print(Car.__dict__['model'])
```

Output:

```
<property object at 0x...>
```

So, in simple words:

Name	Belongs to	What it stores
`model` (in class)	Class-level	A **property object**
`__model` (in instance)	Object-level	The **actual data** stored in memory

---

🧩 What happens when you call `my_car.model`?

Python checks attributes in this order:

1. Looks in the **instance dictionary** (`my_car.__dict__`)
2. If not found, looks in the **class dictionary** (`Car.__dict__`)

When it finds `model` in the class as a **property**, it knows:

> "Ah, this is a *descriptor*, so I must call its `__get__()` method."

That's why this line:

```
print(my_car.model)
```

actually calls:

```
Car.model.__get__(my_car)
```

And that internally runs your getter function:

```
def model(self):
    return self.__model
```

🚫 Why no clash

✅ Class-level name: `model` → refers to the *property descriptor*
✅ Instance-level name: `__model` → refers to the *actual data*

They are totally separate.
You can safely use the same “public name” (`model`) for property, because internally, the real attribute is stored as a *different private name* (`__model`).

⚙️ Analogy

Think of it like this:

Layer	Name you see	What it actually points to
Class layer	`model`	property wrapper
Object layer	`__model`	actual value (like `"Safari"`)

The `property` is just a *smart gatekeeper* between your code and the real data.

Perfect 🔥 — let’s peel this open like a Python autopsy 🧠🐍
We’ll see **exactly** what happens — line by line, inside memory — when you write this:

```
my_car.model = "Nexon"
```

⚙️ Setup (our class)

```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand
        self.__model = model

    @property
    def model(self):
```

```
        return self.__model

    @model.setter
    def model(self, new_model):
        if len(new_model) < 3:
            raise ValueError("Model name too short!")
        print(f"Changing model from {self.__model} → {new_model}")
        self.__model = new_model
```

Then:

```
my_car = Car("Tata", "Safari")
my_car.model = "Nexon"
```

🧩 Step-by-Step Internal Flow

🪞 Step 1 — Instance Creation

When you run:

```
my_car = Car("Tata", "Safari")
```

✅ Memory created for `my_car` object.

Inside it:

```
my_car.__dict__ = {
    "_Car__brand": "Tata",
    "_Car__model": "Safari"
}
```

> ⚠️ Notice the variable is stored as `_Car__model`, because of *name mangling* (from `__model`).

⚙️ Step 2 — You run `my_car.model = "Nexon"`

Here's what Python does **internally**:

(a) Python looks for `"model"` in the instance dictionary:

- Checks `my_car.__dict__` — ❌ Not found (there is only `_Car__model`)
- So it looks into the class `Car.__dict__`

(b) Finds `model` in the class as a property object:

```
print(Car.__dict__['model'])
# <property object at 0x...>
```

This object has three methods internally:
`__get__`, `__set__`, `__delete__` — implemented by the property decorator.

🧠 Step 3 — Python calls the property's `__set__()` method

Instead of directly setting `my_car.model`, Python calls:

```
Car.model.__set__(my_car, "Nexon")
```

Under the hood, that executes your setter function:

```
def model(self, new_model):
    if len(new_model) < 3:
        raise ValueError("Model name too short!")
    print(f"Changing model from {self.__model} → {new_model}")
    self.__model = new_model
```

⚙️ Step 4 — Inside the setter

- It prints: `Changing model from Safari → Nexon`

Then it sets:

`self.__model = new_model`
which means:

`my_car.__dict__["_Car__model"] = "Nexon"`

-

So now the object's internal state is:

```
my_car.__dict__ = {
    "_Car__brand": "Tata",
    "_Car__model": "Nexon"
}
```

✅ No variable named `model` ever gets created.
✅ The `property` only acts as a *controller* for access.

🧠 Step 5 — You print `my_car.model`

Python again looks:

- No `model` in instance dict → finds the **property** in the class.

Calls:

`Car.model.__get__(my_car)`

-

Which runs your getter:

`def model(self): return self.__model`

-

→ Returns `"Nexon"`

🪄 Visualization Summary

Expression	What happens internally	Description
`my_car.model`	`Car.model.__get__(my_car)`	Calls getter
`my_car.model = "Nexon"`	`Car.model.__set__(my_car, "Nexon")`	Calls setter
`del my_car.model`	`Car.model.__delete__(my_car)`	Calls deleter

🎯 Key Takeaway

✅ The “clash” never happens because:

- The *class-level name* (`model`) is a `property` descriptor object.
- The *instance-level data* is stored as a private variable (`_Car__model`).

The property just **controls access** to it — intercepting every read, write, or delete.