🧩 Problem

Modify the Car class so that:

The brand attribute is private (cannot be accessed directly from outside the class).

You provide a getter method to safely access it.

✅ Solution

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

    # Getter method
    def get_brand(self):
        return self.__brand

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


# Create an object
my_car = Car("Toyota", "Corolla")

# Accessing private attribute directly — ❌ Not allowed
# print(my_car.__brand)  # This will cause an AttributeError

# Access via getter — ✅ Correct way
print("Car Brand:", my_car.get_brand())
print("Full Name:", my_car.full_name())
```

🔍 Explanation

Concept	Meaning
self.__brand	The double underscore (__) makes the variable private → can't be accessed directly outside the class.
Getter method	Used to safely retrieve private data without exposing the variable directly.
my_car.get_brand()	Returns the value of __brand safely.

⚠️ If you try to access it directly

```
print(my_car.__brand)
```

you'll get:

```
AttributeError: 'Car' object has no attribute '__brand'
```

🧠 But here’s something cool:

Python doesn’t have true private variables — it uses name mangling.
So internally, __brand is stored as _Car__brand.

You could (but shouldn’t) access it like:

```
print(my_car._Car__brand)
```

This works — but it’s considered bad practice, because it breaks encapsulation.

✅ In short:
Encapsulation = hiding data + controlled access.

🧩 Problem

We want the `brand` attribute to be **private**, but still be readable like this:

```
print(my_car.brand)
```

instead of:

```
print(my_car.get_brand())
```

✅ Pythonic Solution using `@property`

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

    @property
    def brand(self):
        """Getter for brand"""
        return self.__brand

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


# Create object
my_car = Car("Toyota", "Corolla")

# Access brand as if it's public (but it's still private internally!)
print("Car Brand:", my_car.brand)
print("Full Name:", my_car.full_name())
```

🔍 Explanation

Line	What it does
`self.__brand`	Marks the attribute as private.
`@property`	Turns the method below it into a **getter** that acts like an attribute.
`def brand(self):`	This function now runs *whenever you access* `my_car.brand`.
`my_car.brand`	Looks like direct access, but it actually calls the `brand()` method behind the scenes.

⚙️ Bonus: Add a Setter

You can even allow **controlled modification** of private data using `@<property>.setter`:

```
class Car:
    def __init__(self, brand, model):
        self.__brand = brand
        self.model = model
    @property
    def brand(self):
        return self.__brand

    @brand.setter
    def brand(self, new_brand):
        if new_brand:  # simple validation
            self.__brand = new_brand
        else:
            print("❌ Brand name cannot be empty")


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

print(my_car.brand)  # calls getter

my_car.brand = "Mahindra"  # calls setter
print(my_car.brand)

my_car.brand = ""  # invalid brand
```

🧠 Output

```
Tata
Mahindra
❌ Brand name cannot be empty
```

✅ Summary:

Feature	Without `@property`	With `@property`
Read attribute	`obj.get_brand()`	`obj.brand`

Modify attribute	`obj.set_brand('x')`	`obj.brand = 'x'`
Clean code	❌	✅ Very clean
Encapsulation	✅	✅

🧩 Example Recap

Let's take this version of your class 👇

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

    @property
    def brand(self):
        return self.__brand

    @brand.setter
    def brand(self, new_brand):
        if new_brand:
            self.__brand = new_brand
        else:
            print("❌ Brand name cannot be empty")
```

⚙️ Visual Flow of How `@property` Works

```
┌──────────────────────────────┐
│         my_car.brand         │
└───────────────┬──────────────┘
                │
                ▼
       Calls getter method:
```

```
┌──────────────────────────────┐
|  def brand(self):            |
|      return self.__brand     |
└──────────────────────────────┘
              |
              ▼
  Returns private variable
     self.__brand
```

🧩 When you read the value:

```
print(my_car.brand)
```

➡️ Internally Python does:

```
Car.brand.__get__(my_car)
```

This means it **calls the method decorated with `@property`** (the getter) and returns its value.

🧩 When you modify the value:

```
my_car.brand = "Mahindra"
```

➡️ Internally Python does:

```
Car.brand.__set__(my_car, "Mahindra")
```

So it actually **calls the method decorated with `@brand.setter`** and passes `"Mahindra"` as `new_brand`.

🧠 Internal Chain

```
my_car.brand → calls → brand() (getter)
my_car.brand = "Tata" → calls → brand() (setter)
```

Behind the scenes:

- The getter runs when you **access** the property.
- The setter runs when you **assign** a new value.
- Both internally work with a **private variable** (here, `__brand`).

🧩 Visualization Summary

Action	What You Write	What Actually Happens Internally
Read	`my_car.brand`	Calls `Car.brand.__get__(my_car)`
Write	`my_car.brand = "Tata"`	Calls `Car.brand.__set__(my_car, "Tata")`
Delete	`del my_car.brand`	Calls `Car.brand.__delete__(my_car)` *(if defined)*

So, `@property` acts like a **smart wrapper** —
you use it like a normal variable,
but it secretly runs methods to validate or process data.