🧩 The code again

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

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

🧠 Step 1: How Python calls methods

When you run:

```
my_car.full_name()
```

Python automatically translates it behind the scenes to:

```
Car.full_name(my_car)
```

So the method `full_name()` always receives the **object itself** as the **first argument**.

That's what `self` is — it's a variable that **points to the current object**.

🧩 Step 2: Why we write `self.brand`

Each object of the class has its **own** copy of data:

```
car1 = Car("Tata", "Safari")
car2 = Car("Honda", "City")
```

So we have in memory:

Object	brand	model
`car1`	Tata	Safari
`car2`	Honda	City

Now, inside `full_name()`, how do we tell Python *which* car's brand and model to use?

We use `self`.

When you call:

```
car1.full_name()
```

Python runs:

```
Car.full_name(car1)
```

Inside the method:

- `self` → `car1`
- So `self.brand` → `car1.brand` → `"Tata"`
- `self.model` → `car1.model` → `"Safari"`

That's why `return f"{self.brand} {self.model}"` prints:

```
Tata Safari
```

🚫 If you remove `self`

Suppose you wrote:

```
def full_name():
    return f"{brand} {model}"
```

Then Python would complain:

```
TypeError: full_name() takes 0 positional arguments but 1 was given
```

Because it still passes the object automatically (as `self`), but your function doesn't expect it.

Even if it didn't error, the function wouldn't know *which car's brand and model* to use — those are stored **inside each object**, not globally.

🧠 In short

Concept	Meaning
`self`	Refers to the current object being used
`self.brand`	Accesses that object's data
Needed because	Python automatically passes the object as first argument
Without `self`	Function won't know which object's data to use