

Explanation

Concept	Description
<code>class ElectricCar(Car)</code>	ElectricCar <i>inherits</i> from Car
<code>super().__init__(brand, model)</code>	Calls parent class's constructor so we don't rewrite the same code
<code>self.battery_size</code>	Adds a new attribute specific to ElectricCar
<code>my_tesla.full_name()</code>	Works because ElectricCar inherits methods from Car

How inheritance works (visually)

```
Car
├── brand
└── model
└── full_name()

ElectricCar (inherits from Car)
├── battery_size
└── battery_info()
```

So, `ElectricCar` gets everything from `Car` **plus** its own new features.

Quick Test

```
print(isinstance(my_tesla, ElectricCar)) # True
print(isinstance(my_tesla, Car))          # True
```

 Because `ElectricCar` *is-a* `Car`.

what is wrong here

```
class Car:  
    def __init__(self,brand,model):  
        self.brand = brand  
        self.model = model  
  
    def full_name(self):  
        return f"{self.brand} and {self.model}"  
  
  
  
class ElectricCar(Car):  
    def __init__(self,brand,model,battery_size):  
        self.battery_size = battery_size  
        super().__init__(brand,model)  
  
    def battery_size(self):  
        return f"Battery Size - {self.battery_size}"  
  
  
  
my_tesla = ElectricCar("Tesla","Model S","85 kWh")  
  
print(my_tesla.full_name())  
print(my_tesla.battery_size())
```

You wrote:

```
self.battery_size = battery_size
```

and later:

```
def battery_size(self):  
    return f"Battery Size - {self.battery_size}"
```

That means:

- `self.battery_size` → is an **attribute** (a string like "85 kWh")

- `battery_size()` → is a **method** (a function)

Both have the **same name**, and that confuses Python.

What actually happens in memory

When you create:

```
my_tesla = ElectricCar("Tesla", "Model S", "85 kWh")
```

Python executes this line in order:

```
self.battery_size = battery_size
```

Now, inside the object:

```
battery_size = "85 kWh"
```

Later, when you define:

```
def battery_size(self):  
    ...
```

that name `battery_size` (function) gets **overridden** by the attribute value in the instance.

So when you call:

```
my_tesla.battery_size()
```

Python looks up `battery_size` on the object first — finds the **string**, not the method.

Hence it tries to call `"85 kWh"()` which leads to:

```
TypeError: 'str' object is not callable
```

The fix

Just **rename the method** to something else, like `battery_info`:

```
class Car:  
    def __init__(self, brand, model):  
        self.brand = brand  
        self.model = model  
  
    def full_name(self):  
        return f"{self.brand} and {self.model}"  
  
  
class ElectricCar(Car):  
    def __init__(self, brand, model, battery_size):  
        super().__init__(brand, model)  
        self.battery_size = battery_size  
  
    def battery_info(self):  
        return f"Battery Size - {self.battery_size}"  
  
  
my_tesla = ElectricCar("Tesla", "Model S", "85 kWh")  
  
print(my_tesla.full_name())  
print(my_tesla.battery_info())
```

Output:

```
Tesla and Model S  
Battery Size - 85 kWh
```

Summary

Problem

Why it happened

Fix

Attribute & method share the same name

Python treats the attribute as the first match

Rename the method (e.g., `battery_info`)

Order of
`super().__init__()`

Should usually come before using parent attributes

(Optional) Place it first for clarity