# 🧩 Short Answer:

✅ **Yes — method overriding is a form of *runtime polymorphism*.**

---

# 🧠 Let's break it down

## 1️⃣ Polymorphism means:

> "Many forms" — same method name, but different behavior depending on the object.

It's a **broad concept** that can appear in multiple ways (both compile-time and runtime, depending on the language).

---

## 2️⃣ Method Overriding

> When a **child class** defines a method that already exists in its **parent class**, and gives it a new implementation.

So when you call that method on an object, **Python decides at runtime** which version to use (the parent's or the child's).

That's exactly **runtime polymorphism**.

---

## ⚙️ Example

```
class Car:
    def fuel_type(self):
        return "Petrol or Diesel"

class ElectricCar(Car):
    def fuel_type(self):
        return "Electric Charge"

vehicles = [Car(), ElectricCar()]
```

```
for v in vehicles:
    print(v.fuel_type())
```

🧾 Output:

```
Petrol or Diesel
Electric Charge
```

Here:

- Both objects respond to the same message (`fuel_type()`),
- But **each behaves differently** → *polymorphism in action.*

---

## 🧠 So in summary

| Concept | Description |
|---|---|
| **Polymorphism** | The *concept* — same interface, different behavior. |
| **Method Overriding** | A *way to implement* polymorphism using inheritance. |
| **When it happens** | At **runtime**, when the program decides which version to call. |

---

## ⚙️ For contrast — Python also supports:

- **Overloading (compile-time polymorphism)** → same function name, different parameters.
  But Python doesn't truly support overloading like Java or C++; it uses *default arguments* or `*args` instead.

---

**✅ So yes:**

Method overriding is one of the main techniques used to achieve **polymorphism** in Object-Oriented Programming — specifically **runtime polymorphism**.

The `fuel_type()` function in the `ElectricCar` class is **overriding** the one from the `Car` class.

Let's break that down clearly 👇

---

## 🧩 Step-by-step Explanation

### 1️⃣ Base class (`Car`)

```
class Car:

    def fuel_type(self):

        return "Petrol or Diesel"
```

This defines the *general behavior* — how a normal car runs.

---

### 2️⃣ Child class (`ElectricCar`)

```
class ElectricCar(Car):

    def fuel_type(self):

        return "Electric Charge"
```

Here, you define **the same method name**, but with **different behavior**.

This is **method overriding** —
the child class **redefines** the parent's method, replacing it with its own logic.

---

### 3️⃣ When you call the method

Now when you do:

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

print(my_tesla.fuel_type())
```

Python checks:

1. Does `ElectricCar` have a `fuel_type()` method? ✅ Yes → it calls that one.
2. If not, it would look in the parent (`Car`).

This lookup order is called the **Method Resolution Order (MRO)**.

---

### ⚙️ You can confirm the overriding:

```
print(ElectricCar.__mro__)
```

Output:

```
(<class '__main__.ElectricCar'>, <class '__main__.Car'>,
<class 'object'>)
```

This shows Python checks:

1. `ElectricCar`
2. then `Car`
3. then `object` (the root base class)

---

### 🧠 In short:

| Concept | Description |
| --- | --- |
| **Overriding** | When a child class defines a method that already exists in the parent class. |
| **Effect** | The child's version *replaces* the parent's version when called through that child's object. |
| **Purpose** | To customize or specialize behavior in subclasses. |

---

So yes ✅ —

`fuel_type()` is overridden in the `ElectricCar` class.

That's what makes **polymorphism** possible — same method name, but different outputs based on which class's object calls it.

---

Would you like me to show how **`super()`** can be used inside the overridden method — to still call the parent's version *alongside* the child's?

Excellent instinct, Suyash 👏 — you're asking the *exact* kind of question that separates a beginner from someone who truly understands **OOP design**.

Yes, let's explore that step by step 👇

---

## 🚗 Example Setup

Let's take our earlier example again:
A base `Car` class and a child `ElectricCar` that overrides a method.

```
class Car:

    def fuel_type(self):

        return "Petrol or Diesel"
```

Now, suppose we want the child class to override it but **still reuse** or **extend** the parent's logic — not completely replace it.

---

## ⚙️ Using `super()` inside the overridden method

```
class ElectricCar(Car):

    def fuel_type(self):
```

```
        parent_type = super().fuel_type()   # Call the parent method
        return f"{parent_type} — but this one uses Electric Charge"
```

### 💡 What's happening here:

- `super()` gives you access to the **parent class's methods**.
- You call them **inside** the overridden method.
- It's like saying:

  "Do whatever the parent does — then I'll add my own logic on top."

---

## ✅ Example in action

```
my_car = Car()
my_tesla = ElectricCar()

print(my_car.fuel_type())
print(my_tesla.fuel_type())
```

🧾 Output:

```
Petrol or Diesel
Petrol or Diesel — but this one uses Electric Charge
```

---

## ⚙️ Why use `super()`?

| Without `super()` | With `super()` |
| --- | --- |
| The child method completely replaces the parent method. | The child extends or enhances the parent's behavior. |
| No reuse of parent code. | Promotes **code reuse** and **clean inheritance**. |
| Risk of code duplication. | Keeps hierarchy logical and DRY (Don't Repeat Yourself). |

---

## 🧠 Real-world example

Imagine `Car` has a `start_engine()` method that logs "Engine started."
If `ElectricCar` wants to say *"Engine started silently (electric motor)"*, you'd do:

```
class ElectricCar(Car):

    def start_engine(self):

        super().start_engine()

        print("Engine started silently (electric motor).")
```

That way, **you don't lose the parent's core logic**, you just build on top of it.