BOOKLET 7 OF 8

Python Classes, Part 2

Welcome to Session 7! In this booklet, you will learn how to build relationships between classes using **inheritance** and **composition**, and how objects can interact with each other. These are some of the most powerful patterns in object-oriented programming.

1

Inheritance

Parent & child classes

2

super()

Reuse parent setup

3

Overriding

Customize child behavior

4

Composition

Has-a relationships

5

Interaction

Objects calling methods

LEARNING GOALS

What You Will Be Able to Do

By the end of this session, you should be able to accomplish all of the following:

1

Build inheritance

Create a parent class and a child class that inherits from it.

2

Use `super()`

Reuse the parent's initialization code without rewriting it.

3

Override methods

Give a child class its own version of a parent method.

4

Apply composition

Store one class inside another to model "has-a" relationships.

5

Connect objects

Use dot notation like `ev.battery.describe_battery()` to make objects interact.

CONCEPT 1

Inheritance: Parent and Child Classes

The Core Idea

Use inheritance when one class is a **specialized type** of another. The child class gets everything the parent has, and can add more.

- **Parent class** – the general version
- **Child class** – a more specific version

Example: `ElectricVehicle` *is a* `Vehicle`

Parent Class Example

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def describe(self):
        print(f"{self.year} {self.make} {self.model}")
```

`__init__` stores the data. `describe()` prints a simple summary of the vehicle.

CONCEPT 2

Using `super()` to Reuse Parent Setup

When a child class shares attributes with its parent, use `super()` to call the parent's `__init__()` instead of rewriting it.

Child Class with `super()`

```
class ElectricVehicle(Vehicle):
    def __init__(self, make, model,
                 year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity
```

What Each Line Does

- `ElectricVehicle(Vehicle)` – declares inheritance from Vehicle
- `super().__init__(...)` – runs the parent's setup for `make`, `model`, `year`
- `self.battery_capacity` – adds a new attribute only the child has

Because `describe()` is defined in `Vehicle`, child objects can still call it – even without redefining it.

CONCEPT 3

Overriding Methods

Overriding means defining a method in the child class with the **same name** as a parent method. Python will use the child's version for child objects.

Override `describe()` in the Child

```
class ElectricVehicle(Vehicle):
    def __init__(self, make, model,
                 year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity

    def describe(self):
        print(
            f"{self.year} {self.make} "
            f"{self.model} with "
            f"{self.battery_capacity} kWh battery"
        )
```

Key Points

- The child's `describe()` replaces the parent's **only for EV objects**.
- The parent's `describe()` still works for `Vehicle` objects.
- The method name must be spelled **exactly** the same – a typo means no override happens.

Want both? Call `super().describe()` inside the override to run the parent version first.

CONCEPT 4

Composition: "Has-a" Relationships

Use **composition** when one class should *contain* another class as an attribute. A car **has a** battery – it is not a battery.

Step 1 — Define the Contained Class

```
class Battery:
    def __init__(self, capacity):
        self.capacity = capacity

    def describe_battery(self):
        print(
            f"Battery capacity: "
            f"{self.capacity} kWh"
        )
```

Step 2 — Store It Inside the Container

```
class ElectricVehicle:
    def __init__(self, make, model,
                 year, capacity):
        self.make = make
        self.model = model
        self.year = year
        self.battery = Battery(capacity)

    def describe(self):
        print(f"{self.year} {self.make} "
              f"{self.model}")
        self.battery.describe_battery()
```

self.battery stores a whole `Battery` object. When `describe()` runs, it first prints EV info, then asks the battery object to describe itself.

CONCEPT 5

Object Interaction with Dot Notation

When you use composition, objects can "talk" to each other by chaining dot notation. This lets you reach inside one object to call a method on another.

Example

```
ev = ElectricVehicle(
    "Tesla", "Model 3", 2024, 75
)
ev.battery.describe_battery()
```

Output:

```
Battery capacity: 75 kWh
```

Reading the Chain

- `ev` – the ElectricVehicle object
- `ev.battery` – the Battery object stored inside `ev`
- `ev.battery.describe_battery()` – runs Battery's method

CONCEPT 6

Storing Multiple Objects: The Fleet Pattern

A common pattern is a **container class** that holds a list of objects and calls methods on each one. This is a great example of composition at work.

Fleet Class

```
class Fleet:
    def __init__(self):
        self.vehicles = []

    def add_vehicle(self, vehicle):
        self.vehicles.append(vehicle)

    def show_all(self):
        for v in self.vehicles:
            v.describe()
```

How It Works

vehicles

A list that stores any vehicle objects you add.

add_vehicle()

Appends any object that has a describe() method.

show_all()

Loops through the list and calls describe() on each.

Inheritance vs. Composition at a Glance

Choosing between inheritance and composition depends on the relationship between your classes. Use this table as a quick reference:

Pattern	Relationship	Example
Inheritance	"is-a" – child is a type of parent	`ElectricVehicle` is a `Vehicle`
Composition	"has-a" – class contains another	`ElectricVehicle` has a `Battery`
Overriding	Child replaces parent method	Child `describe()` adds battery info
Container class	Holds a collection of objects	`Fleet` stores a list of vehicles

STEP-BY-STEP EXAMPLE

Full Example: Fleet of Vehicles

This complete example brings together inheritance, `super()`, overriding, and a container class. Read it top to bottom, then copy and run it.

```
class Vehicle:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def describe(self):
        print(f"{self.year} {self.make} {self.model}")


class ElectricVehicle(Vehicle):
    def __init__(self, make, model, year, battery_capacity):
        super().__init__(make, model, year)
        self.battery_capacity = battery_capacity

    def describe(self):
        print(f"{self.year} {self.make} {self.model} with {self.battery_capacity} kWh battery")

    def charge(self):
        print(f"Charging {self.make} {self.model}...")


class GasolineVehicle(Vehicle):
    def __init__(self, make, model, year, tank_size):
        super().__init__(make, model, year)
        self.tank_size = tank_size

    def describe(self):
        print(f"{self.year} {self.make} {self.model} with {self.tank_size} L tank")

    def refuel(self):
        print(f"Refueling {self.make} {self.model}...")


class Fleet:
    def __init__(self):
        self.vehicles = []

    def add_vehicle(self, vehicle):
        self.vehicles.append(vehicle)

    def show_all(self):
        for v in self.vehicles:
            v.describe()
```

STEP-BY-STEP EXAMPLE (CONTINUED)

Creating Objects and Running the Fleet

Add this code below the class definitions to create objects, call methods, and use the `Fleet`:

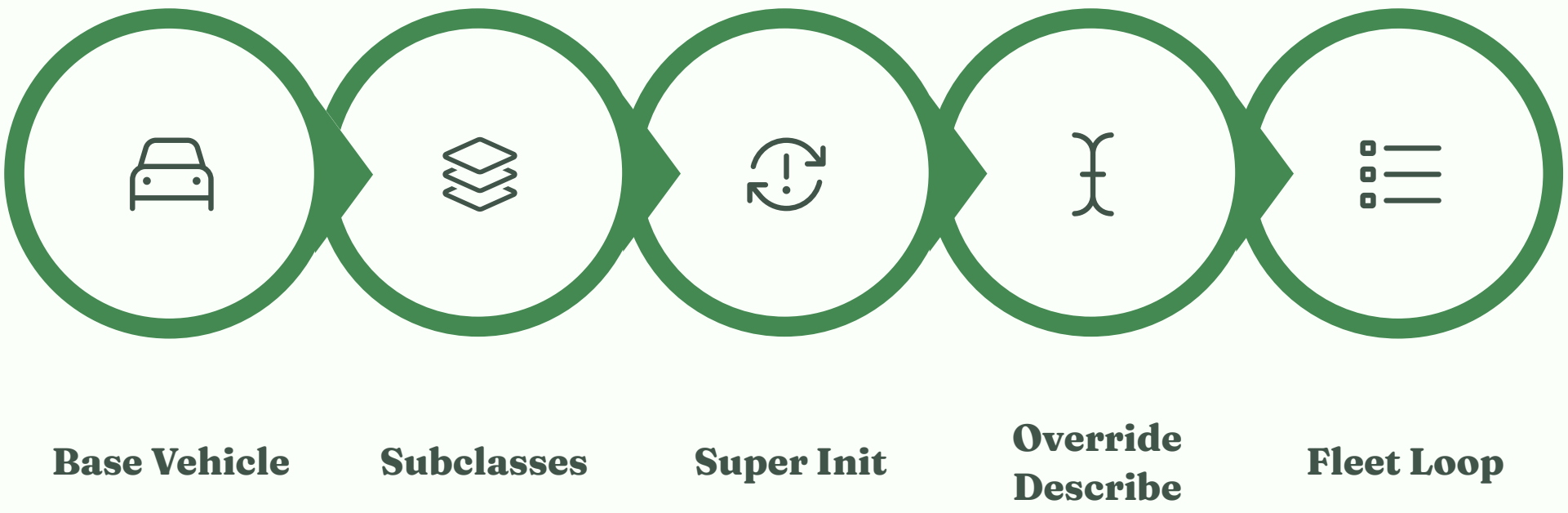
```
# Create objects
v1  = Vehicle("CityMotors", "Comet", 2010)
ev1 = ElectricVehicle("Electra", "Spark", 2024, 75)
gv1 = GasolineVehicle("RoadCo", "Ranger", 2018, 50)

# Call individual methods
v1.describe()
ev1.describe()
gv1.describe()

ev1.charge()
gv1.refuel()

# Add to fleet and display all
fleet = Fleet()
fleet.add_vehicle(v1)
fleet.add_vehicle(ev1)
fleet.add_vehicle(gv1)

print("\nFleet overview:")
fleet.show_all()
```



STEP-BY-STEP LOGIC

How the Full Example Works

1 `Vehicle` defines the foundation

Shared attributes (`make`, `model`, `year`) and a shared `describe()` method live here.

2 Child classes inherit from `Vehicle`

Both `ElectricVehicle` and `GasolineVehicle` extend the parent with their own extra attributes.

3 `super().__init__()` handles shared setup

Each child calls `super()` so `make`, `model`, and `year` are always created correctly.

4 Overridden `describe()` adds detail

Each child's version of `describe()` prints extra info – battery kWh or tank liters.

5 `Fleet` unifies everything

The container stores mixed vehicle types in one list and loops through calling `describe()` on each.

⚠️ COMMON MISTAKES

5 Mistakes to Avoid

Forgetting `super().__init__()`

Without it, parent attributes like `make` and `year` do not exist on the child object – you will get an `AttributeError`.

Wrong indentation inside a class

Methods must be indented inside the class body. If not, Python may treat them as standalone functions or raise an error.

Typo in an overriding method name

Writing `desribe` instead of `describe` creates a *new* method – the parent version is used instead, which is not the intended behavior.

Forgetting to create the composed object

Calling `self.battery.describe_battery()` without first doing `self.battery = Battery(...)` causes an `AttributeError`.

Missing the correct prefix on composed attributes

Use `self.battery.capacity`, not just `capacity` – otherwise Python looks for a local variable that does not exist.

PRACTICE TASKS

Practice Task 1: GasolineVehicle Subclass

Your Task

Create a `GasolineVehicle` class that inherits from `Vehicle`.

- Add one new attribute: `tank_size`
- Use `super()` for the shared attributes (`make`, `model`, `year`)
- Test it by creating a `GasolineVehicle` object and calling `describe()`

Starter Structure

```
class GasolineVehicle(Vehicle):
    def __init__(self, make, model,
                 year, tank_size):
        super().__init__(???)
        self.tank_size = ???
```

Replace the `???` with the correct values.

PRACTICE TASKS

Practice Task 2: Override `describe()`

Your Task

Override `describe()` in `GasolineVehicle` so it prints tank size as well as make, model, and year.

Expected output example:
`2018 RoadCo Ranger with 50 L tank`

Hint

```
def describe(self):
    print(
        f"{self.year} {self.make} "
        f"{self.model} with "
        f"{self.tank_size} L tank"
    )
```

Add this method **inside** the `GasolineVehicle` class, indented correctly.

PRACTICE TASKS

Practice Task 3: Add Custom Methods

Add unique methods to each child class to model behavior that only that type of vehicle can do.

ElectricVehicle — Add `charge()`

```
def charge(self):
    print(
        f"Charging "
        f"{self.make} {self.model}..."
    )
```

Create an `ElectricVehicle` object and call `charge()` on it.

GasolineVehicle — Add `refuel()`

```
def refuel(self):
    print(
        f"Refueling "
        f"{self.make} {self.model}..."
    )
```

Create a `GasolineVehicle` object and call `refuel()` on it.

These methods only exist on their respective child classes. Calling `charge()` on a plain `Vehicle` would raise an `AttributeError`.

PRACTICE TASKS

Practice Task 4: Add Composition with Battery

Use composition to give every `ElectricVehicle` its own `Battery` object automatically.

Step 1 — Create the `Battery` Class

```
class Battery:
    def __init__(self, capacity):
        self.capacity = capacity

    def show_range(self):
        estimated = self.capacity * 5
        print(
            f"Estimated range: "
            f"{estimated} km"
        )
```

Step 2 — Compose It Inside EV

```
class ElectricVehicle(Vehicle):
    def __init__(self, make, model,
                 year, capacity):
        super().__init__(make, model, year)
        self.battery = Battery(capacity)

# Test it:
ev = ElectricVehicle(
    "Electra", "Spark", 2024, 75
)
ev.battery.show_range()
```

PRACTICE TASKS

Practice Task 5: Create a Fleet

Your Task

Build a `Fleet` class that stores multiple vehicles and can display them all.

- Use a list to store vehicles
- Add an `add_vehicle()` method
- Add a `show_all()` method that calls `describe()` on each
- Add at least one `Vehicle`, one `ElectricVehicle`, and one `GasolineVehicle` to your fleet

Expected Pattern

```
fleet = Fleet()
fleet.add_vehicle(v1)
fleet.add_vehicle(ev1)
fleet.add_vehicle(gv1)
fleet.show_all()

# Output (example):
# 2010 CityMotors Comet
# 2024 Electra Spark with 75 kWh battery
# 2018 RoadCo Ranger with 50 L tank
```

Each vehicle's overridden `describe()` produces its own custom output.

EXPLORATION QUESTIONS

Think Deeper: Exploration Questions

These questions are designed to push your understanding further. Try them in your editor before reading ahead.

1

Question 1

Remove `super().__init__(make, model, year)` from a child class. What error appears when you call `describe()`? Why does that error happen?

2

Question 2

In the `Fleet` class, what happens if you add an object that does **not** have a `describe()` method? What error message do you see?

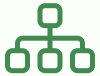
3

Question 3

Inside `ElectricVehicle.describe()`, add a call to `super().describe()` at the top. How does the printed output change?

SESSION SUMMARY

What You Learned in Session 7



Inheritance

Reuse code when classes share an "is-a" relationship. Child classes get everything from the parent and can add more.

`super()`

Calls the parent's `__init__()` from the child so you never rewrite shared setup code.

Overriding

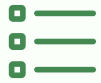
Define a method in the child with the same name as the parent. Python uses the child's version for child objects.

Composition

Model "has-a" relationships by storing one class inside another as an attribute.

Object Interaction

Use dot notation like `ev.battery.describe_battery()` to reach inside one object and call a method on another.



Container Classes

A class like `Fleet` can store many objects in a list and loop through them to call shared methods.

Up Next: Session 8

You have completed 7 out of 8 sessions. In the final session, you will build on everything you have learned so far.

Sessions 1–3

Variables, data types, control flow

Sessions 4–5

Functions, modules, file handling

Sessions 6–7

Classes Part 1 & 2 – OOP foundations

Session 8 →

Putting it all together – final project

Make sure you are comfortable with all five concepts from this session before moving on. The final session builds directly on inheritance and composition.