Composition in Python

What Is Composition?

Composition is an Object Oriented Programming technique where one class **contains an object of another class** and uses it to perform part of its work.

The simplest way to remember it is:

> **Composition means HAS A.**

For example:

```
Car HAS an Engine
Student HAS an Address
Order HAS a Customer
Computer HAS a Keyboard
```

The objects work together, but one object does not necessarily become a specialized version of the other.

Composition vs Inheritance

This is one of the most important OOP design decisions.

Inheritance

Inheritance represents:

>

> **IS A**

`Dog IS an Animal`
`PDFResource IS a Resource`
`Manager IS an Employee`

Example:

```
class Animal:
    pass


class Dog(Animal):
    pass
```

`Dog` is a type of `Animal`.

Composition

Composition represents:

> **HAS A**

`Car HAS an Engine`

Example:

```
class Engine:
    pass
```

```
class Car:
    def __init__(self):
        self.engine = Engine()
```

A `Car` is not an `Engine`.

It **has an Engine**.

Why Does Composition Matter?

Large applications are made from many different components.

Instead of creating one huge class that does everything, we can divide responsibilities between smaller classes and make them work together.

For example, instead of:

```
class OnlineStore:
    # customer logic
    # payment logic
    # order logic
    # email logic
    # inventory logic
```

we can create separate components:

```
OnlineStore
□□□ Customer
□□□ Order
□□□ Payment
```

□□□ Inventory
□□□ Notification

Then the main object can **compose** these components.

This makes the code easier to understand, test, change, and extend.

A Simple Example

Let's create an `Engine`:

```
class Engine:
    def start(self):
        print("Engine started")
```

Now create a `Car`:

```
class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()
        print("Car started")
```

Create the object:

```
car = Car()
car.start()
```

Output:

```
Engine started
Car started
```

The `Car` delegates engine-related work to its `Engine` object.

Understanding `self.engine`

This line is the key:

```
self.engine = Engine()
```

It means:

> "Every `Car` object has an `Engine` object."

For example:

```
car = Car()
```

Conceptually:

```
car
 □
 □□□ engine
     □
     □□□ Engine object
```

The `Car` can now use:

```
self.engine.start()
```

This relationship is composition.

Passing Dependencies Into a Class

Instead of creating the component inside the class, we can pass it in.

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

    def start(self):
        self.engine.start()
```

Now:

```
engine = Engine()
car = Car(engine)

car.start()
```

This is often called **dependency injection**.

The `Car` does not need to know how to create an engine.

Someone else provides the engine.

Why Dependency Injection Is Useful

Suppose we have:

```
class PetrolEngine:
    def start(self):
        print("Petrol engine started")
```

and:

```
class ElectricMotor:
    def start(self):
        print("Electric motor started")
```

Our `Car` can work with either:

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

    def start(self):
        self.engine.start()
```

Now:

```
petrol_car = Car(PetrolEngine())
electric_car = Car(ElectricMotor())

petrol_car.start()
electric_car.start()
```

Output:

```
Petrol engine started
Electric motor started
```

The `Car` does not need to change.

This combines **composition** with **polymorphism**.

Composition and Polymorphism Together

Composition becomes especially powerful when the composed object follows a common interface.

For example:

```
class EmailSender:
    def send(self, message):
        print(f"Email: {message}")


class SMSSender:
    def send(self, message):
        print(f"SMS: {message}")
```

Now:

```
class NotificationService:
    def __init__(self, sender):
        self.sender = sender

    def notify(self, message):
        self.sender.send(message)
```

We can choose the implementation:

```
email_service = NotificationService(EmailSender())
```

```
sms_service = NotificationService(SMSSender())

email_service.notify("Welcome!")
sms_service.notify("Welcome!")
```

Output:

```
Email: Welcome!
SMS: Welcome!
```

The `NotificationService` **has a sender**.

The sender can have different implementations.

This is a common real-world software design pattern.

A haas.dev Example

Imagine a `Resource` on haas.dev.

A resource might contain:

- title
- author information
- category
- content
- download information

Instead of putting everything into one giant class, we can create smaller classes.

```
class Author:
    def __init__(self, name):
        self.name = name
```

Then:

```
class Resource:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

Create the objects:

```
author = Author("Hafsa")
resource = Resource("Python OOP Guide", author)
```

Now:

```
print(resource.title)
print(resource.author.name)
```

Output:

```
Python OOP Guide
Hafsa
```

The relationship is:

```
Resource
   
   HAS AN
        
      Author
```

The `Resource` does not inherit from `Author`.

It contains an `Author` object.

Composition With Multiple Components

A class can contain several different objects.

For example:

```
class CPU:
    def process(self):
        print("CPU processing")


class Memory:
    def load(self):
        print("Memory loading")


class Storage:
    def read(self):
        print("Storage reading")
```

Now:

```
class Computer:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.storage = Storage()
```

```
    def start(self):
        self.cpu.process()
        self.memory.load()
        self.storage.read()
```

The structure becomes:

```
Computer
□□□ CPU
□□□ Memory
□□□ Storage
```

Each class has a focused responsibility.

Delegation

Composition often uses **delegation**.

Delegation means:

> One object asks another object to perform a task.

Example:

```
class Engine:
    def start(self):
        print("Engine started")
```

The car delegates engine work:

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

    def start(self):
        self.engine.start()
```

The `Car` does not implement the engine's internal behavior.

It simply says:

```
self.engine.start()
```

This separation of responsibility is one of the main benefits of composition.

Composition Reduces Large Classes

Consider this class:

```
class ECommerceSystem:
    def create_order(self):
        ...

    def process_payment(self):
        ...

    def send_email(self):
        ...
```

```
    def update_inventory(self):
        ...

    def generate_invoice(self):
        ...
```

This class has many responsibilities.

Instead, we could create:

```
OrderService
PaymentService
EmailService
InventoryService
InvoiceService
```

Then compose them:

```
class ECommerceSystem:
    def __init__(
        self,
        order_service,
        payment_service,
        email_service,
        inventory_service,
        invoice_service
    ):
        self.orders = order_service
        self.payment = payment_service
        self.email = email_service
        self.inventory = inventory_service
        self.invoice = invoice_service
```

Now each component has a focused responsibility.

Composition vs Inheritance Example

Suppose we have:

```
Vehicle
  
Car
```

Inheritance makes sense because:

```
Car IS A Vehicle
```

But suppose:

```
Car  Engine
```

Inheritance would be wrong:

```
class Car(Engine):
    pass
```

A car is not an engine.

Composition is appropriate:

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

Because:

```
Car HAS AN Engine
```

Inheritance Can Become Rigid

Suppose you build:

```
Vehicle
□□□ PetrolCar
□□□ ElectricCar
□□□ HybridCar
```

As requirements grow, inheritance hierarchies can become complicated.

Composition can allow independent components:

```
Car
□□□ Engine
□□□ Battery
□□□ Navigation
□□□ PaymentSystem
```

Components can potentially be replaced independently.

This often makes systems more flexible.

Composition Over Inheritance

You may hear the phrase:

>

> **Favor composition over inheritance.**

This does **not** mean:

> "Never use inheritance."

It means:

> When both approaches could work, consider whether composing smaller objects gives you a simpler and more flexible design.

Inheritance is useful when there is a genuine **IS-A** relationship.

Composition is useful when an object needs another object to perform part of its responsibilities.

Composition vs Aggregation

You may encounter another OOP term: **aggregation**.

Both involve objects containing or referring to other objects.

A simplified distinction is:

Composition

The contained object is strongly associated with the containing object.

Example:

`House HAS Rooms`

The room is considered part of the house structure.

Aggregation

The objects can exist independently.

Example:

`Department HAS Employees`

An employee can exist even if the department changes.

In Python, there is no special syntax that automatically marks a relationship as composition or aggregation.

The distinction is mainly about the **design relationship and object lifecycle**.

Composition and Object Lifetime

Consider:

```
class Engine:
    pass


class Car:
    def __init__(self):
```

```
        self.engine = Engine()
```

The engine is created as part of creating the car.

Conceptually:

```
Car created
    
Engine created
```

But if we pass an existing object:

```
engine = Engine()
car = Car(engine)
```

the engine can exist independently.

This is why dependency injection often gives you more control over object lifetimes.

Composition Makes Testing Easier

Suppose:

```
class PaymentService:
    def charge(self, amount):
        print(f"Charging {amount}")
```

And:

```
class Checkout:
    def __init__(self, payment_service):
```

```
        self.payment_service = payment_service

    def checkout(self, amount):
        self.payment_service.charge(amount)
```

For testing, we can provide a fake payment service:

```
class FakePaymentService:
    def charge(self, amount):
        print(f"Fake payment: {amount}")
```

Then:

```
checkout = Checkout(FakePaymentService())
checkout.checkout(100)
```

We can test the `Checkout` logic without using a real payment system.

This is one reason dependency injection and composition are common in professional software.

Composition and Encapsulation

Composition also works well with encapsulation.

For example:

```
class Wallet:
    def __init__(self, balance):
        self._balance = balance

    def get_balance(self):
```

```
        return self._balance
```

Now another class can use it:

```
class Customer:
    def __init__(self, wallet):
        self.wallet = wallet
```

The `Customer` does not need to know how the wallet manages its balance.

It simply interacts with the wallet's public behavior.

This creates separation between components.

A More Realistic Example

Consider a task application.

We could have:

```
class User:
    def __init__(self, name):
        self.name = name
```

A task:

```
class Task:
    def __init__(self, title):
        self.title = title
```

And a task manager:

```
class TaskManager:
    def __init__(self, user):
        self.user = user
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)
```

Now:

```
user = User("Hafsa")
manager = TaskManager(user)

manager.add_task(Task("Finish Python OOP guide"))
manager.add_task(Task("Review resources"))
```

The `TaskManager` contains:

```
User
Tasks
```

It composes objects rather than inheriting from them.

A Useful Design Question

When deciding between inheritance and composition, ask:

Question 1

Can I naturally say:

>

> "X is a Y"?

If yes, inheritance might make sense.

Example:

`Dog is an Animal`

Question 2

Can I naturally say:

> "X has a Y"?

If yes, composition might make sense.

Example:

`Car has an Engine`

Question 3

Does X simply need Y's functionality?

If yes, composition is often worth considering.

Composition Mental Model

Think of an application as LEGO blocks.

Instead of building one giant block:

```
ONE HUGE CLASS
```

you build smaller blocks:

```
Authentication
Payment
Database
Notifications
Logging
Storage
```

Then connect them:

```
Application
   
   ├── Authentication
   ├── Payment
   ├── Database
   ├── Notifications
   └── Storage
```

Each component performs a specific job.

The application composes them into a working system.

Common Mistake: Using Inheritance Just for Code Reuse

Suppose you have:

```
class Database:
    def connect(self):
        print("Connected")
```

You might be tempted to write:

```
class UserService(Database):
    pass
```

But a `UserService` is not a `Database`.

If the service needs a database, use composition:

```
class UserService:
    def __init__(self, database):
        self.database = database
```

Now the relationship is correct:

```
UserService HAS A Database
```

Common Mistake: Making Classes Too Dependent

This design:

```
class Checkout:
    def __init__(self):
        self.payment = StripePayment()
```

makes `Checkout` directly dependent on one specific implementation.

A more flexible design:

```
class Checkout:
    def __init__(self, payment):
        self.payment = payment
```

Now another compatible payment implementation can be supplied.

This is an important benefit of dependency injection.

Common Mistake: Creating Composition Without Clear Responsibilities

Composition does not automatically make code better.

Avoid creating tiny classes for every single line of logic.

For example:

```
NameManager
StringManager
NumberManager
AdditionManager
```

if these classes do not represent meaningful responsibilities.

Good composition uses components that represent real concepts or responsibilities.

Common Mistake: Exposing Everything

Suppose:

```
class BankAccount:
    def __init__(self):
        self.database = Database()
        self.payment = Payment()
```

If every part of the system directly modifies every internal component, the classes become tightly coupled.

Prefer clear public methods and boundaries:

```
account.deposit(100)
account.withdraw(50)
```

rather than allowing unrelated code to manipulate internal state everywhere.

Composition With Polymorphism

One of the most powerful combinations is:

```
Composition
    +
Polymorphism
```

For example:

```
class PDFExporter:
    def export(self, data):
```

```
        print("Exporting PDF")


class JSONExporter:
    def export(self, data):
        print("Exporting JSON")
```

Then:

```
class Report:
    def __init__(self, exporter):
        self.exporter = exporter

    def save(self, data):
        self.exporter.export(data)
```

Now:

```
pdf_report = Report(PDFExporter())
json_report = Report(JSONExporter())

pdf_report.save("data")
json_report.save("data")
```

`Report` uses composition.

`PDFExporter` and `JSONExporter` provide polymorphic behavior.

This combination appears frequently in real applications.

Mini Project: Order System

Build an order system using composition.

Create:

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
```

Then:

```
class Cart:
    def __init__(self):
        self.products = []

    def add_product(self, product):
        self.products.append(product)

    def total(self):
        return sum(product.price for product in self.products)
```

Now create:

```
class Order:
    def __init__(self, cart):
        self.cart = cart

    def checkout(self):
        print(f"Total: {self.cart.total()}")
```

Usage:

```
product1 = Product("Python Guide", 100)
```

```
product2 = Product("OOP Guide", 150)

cart = Cart()
cart.add_product(product1)
cart.add_product(product2)

order = Order(cart)
order.checkout()
```

Expected output:

```
Total: 250
```

Relationships:

```
Order
  □
HAS A
  □
Cart
  □
HAS
  □
Products
```

---

5 Minute Challenge

Create:

```
Computer
□□□ CPU
□□□ RAM
```

□□□ Storage

Each component should have one useful method.

For example:

```
cpu.process()
ram.load()
storage.read()
```

Then create:

```
computer.start()
```

The `Computer` should delegate each task to the appropriate component.

Exercises

Exercise 1

Create:

```
Car
Engine
```

The car should receive an engine through its constructor.

Exercise 2

Create:

Student
Address

A student should contain an address object.

Display:

Student name
City
Country

Exercise 3

Create:

Order
Cart
Product

The Order should contain a Cart, and the Cart should contain products.

Calculate the total order price.

Exercise 4

Create:

NotificationService
EmailSender
SMSSender

Use composition so the notification service can work with either sender.

Exercise 5

Create two classes:

`PDFExporter`
`JSONExporter`

Both should provide:

`export(data)`

Create a `Report` class that receives an exporter.

This exercise combines **composition + polymorphism**.

Mini Quiz

1. What does composition usually represent?

A. IS-A
B. HAS-A
C. IS-NOT-A
D. ONLY-A

Answer: B

---

2. Which relationship is better represented by composition?

A. Dog is an Animal
B. Car has an Engine
C. Manager is an Employee
D. Cat is an Animal

Answer: B

---

3. What does this line demonstrate?

```
self.engine = Engine()
```

A. Inheritance
B. Composition
C. Recursion
D. Exception handling

Answer: B

---

4. What is dependency injection?

A. Creating every dependency inside a class
B. Passing required dependencies into a class
C. Inheriting from every dependency
D. Removing all classes

Answer: B

---

5. Why can composition improve flexibility?

A. It eliminates all code
B. Components can be replaced or changed independently
C. It prevents objects from working together
D. It removes the need for methods

Answer: B

Interview Questions

Beginner

1. What is composition in Python?

Composition is an OOP technique where one object contains or uses another object to perform part of its responsibilities.

2. What is the difference between inheritance and composition?

Inheritance represents an **IS-A** relationship, while composition generally represents a **HAS-A** relationship.

3. Give an example of composition.

A `Car` has an `Engine`, so:

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

Intermediate

4. What is dependency injection?

Dependency injection means providing an object with the dependencies it needs instead of forcing it to create those dependencies itself.

5. Why is composition often preferred over inheritance?

Composition can reduce coupling and allow components to be replaced or combined more flexibly.

6. Does composition eliminate inheritance?

No. Both are useful. The correct choice depends on the relationship and design requirements.

Practical

7. How does composition work with polymorphism?

A class can contain an object that follows a common interface while allowing different implementations to be supplied.

Example:

```
class Report:
    def __init__(self, exporter):
        self.exporter = exporter
```

Different exporters can then be injected.

8. Why does dependency injection help testing?

A real dependency can be replaced with a fake or mock implementation, allowing the class to be tested in isolation.

Composition Cheat Sheet

Composition
□
□□□ HAS-A relationship
□
□□□ Objects contain/use other objects
□
□□□ Encourages smaller responsibilities
□
□□□ Supports delegation
□
□□□ Works well with polymorphism
□
□□□ Often improves flexibility

Basic composition

```
class Engine:
    def start(self):
        print("Engine started")


class Car:
    def __init__(self):
        self.engine = Engine()
```

Dependency injection

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

Delegation

```
class Car:
    def start(self):
        self.engine.start()
```

Composition + polymorphism

```
class Report:
    def __init__(self, exporter):
        self.exporter = exporter

    def save(self, data):
        self.exporter.export(data)
```

Quick Comparison

Concept	Main Relationship	Example
Inheritance	IS-A	Dog Animal
Composition	HAS-A	Car Engine
Polymorphism	Same interface, different behavior	`export()`

Encapsulation	Controlled access	`account.deposit()`
Abstraction	Hide unnecessary implementation details	Common interface

---

Key Takeaways

1. **Composition means building objects from other objects.**
2. It usually represents a **HAS-A** relationship.
3. `Car HAS an Engine` is composition.
4. Composition is different from inheritance.
5. Composition allows responsibilities to be divided between smaller classes.
6. **Delegation** allows one object to ask another object to perform a task.
7. **Dependency injection** allows dependencies to be supplied from outside.
8. Composition works especially well with polymorphism.
9. Composition can reduce tight coupling and improve testability.
10. Do not use inheritance simply because it provides code reuse.
11. Use inheritance for meaningful **IS-A** relationships.
12. Use composition when an object **has**, **uses**, or **depends on** another object.

The core idea is:

```
Inheritance:
Dog IS AN Animal

Composition:
Car HAS AN Engine
```

Composition + Polymorphism:
Report HAS AN Exporter
Exporter can be PDF, JSON, CSV, etc.

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app