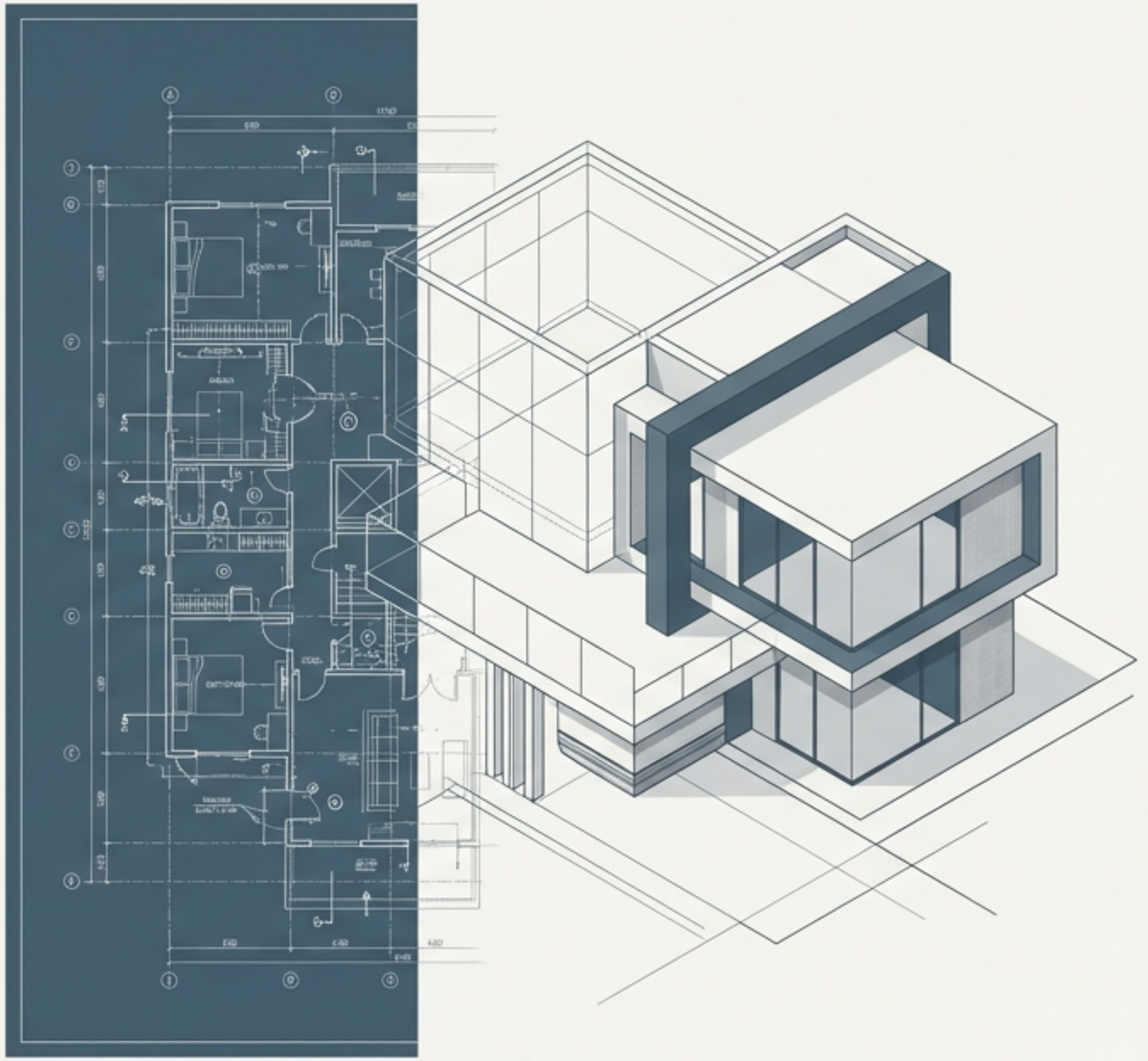
The Object-Oriented Paradigm

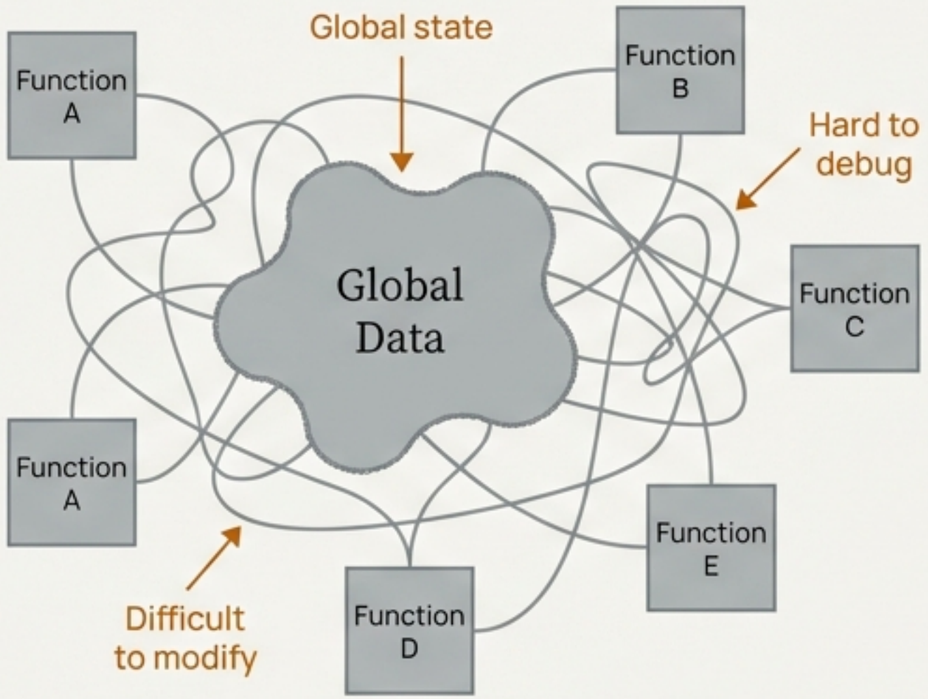
From Data Structures to Digital Models

A guide to structuring your code to mirror the real world, creating software that is more intuitive, manageable, and powerful.

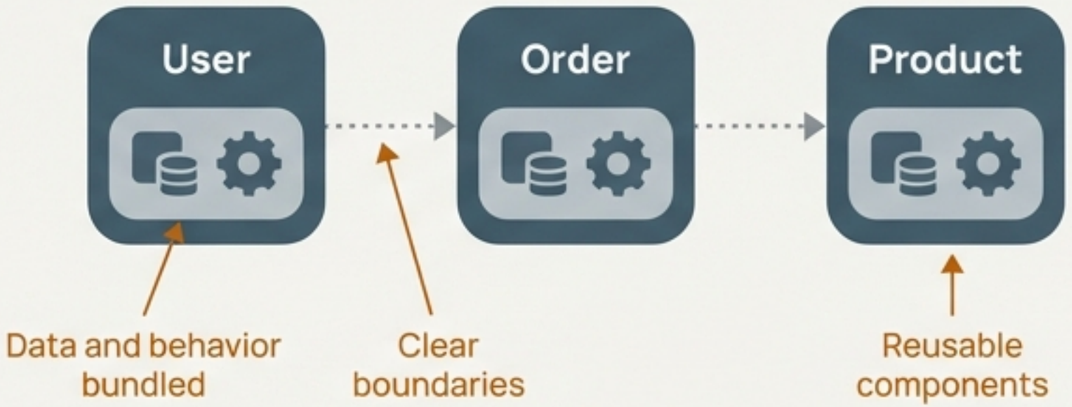


The Shift from Process to Object

A World of Functions



A World of Objects



Procedural programming focuses on a sequence of actions. Object-Oriented Programming focuses on creating self-contained objects that model real-world entities, bundling data and the functions that operate on that data together.

The Core Idea: Blueprints and the Buildings They Create

The Blueprint (Class)

A class is a template for creating objects. It defines a set of attributes (data) and methods (behaviors) that the created objects will have.

```
# The blueprint for all bank accounts
class BankAccount:
    # The constructor: called when a new object is created
    def __init__(self, owner, balance=0):
        self.owner = owner   # Instance attribute
        self.balance = balance # Instance attribute
```

The Blueprint

An Attribute (Data)

The Constructor

The Buildings (Objects)

An object (or instance) is a concrete occurrence of a class. You can create many unique objects from a single class blueprint.

```
# Creating two unique "buildings" from the blueprint
account1 = BankAccount("Alice", 1000)
account2 = BankAccount("Bob", 500)
```

account1 (Owner: Alice, Balance: 1000)

account2 (Owner: Bob, Balance: 500)

Bringing Objects to Life with Methods

Methods are functions that belong to a class. They define the behaviors of an object, allowing it to perform actions and interact with its own data.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    # A method to add funds
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount}. New balance: {self.balance}")

    # A method to remove funds
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrew {amount}. New balance: {self.balance}")
        else:
            print("Insufficient funds.")

    # A method that returns a value
    def get_balance(self):
        return self.balance
```

How It Works

The `self` Parameter

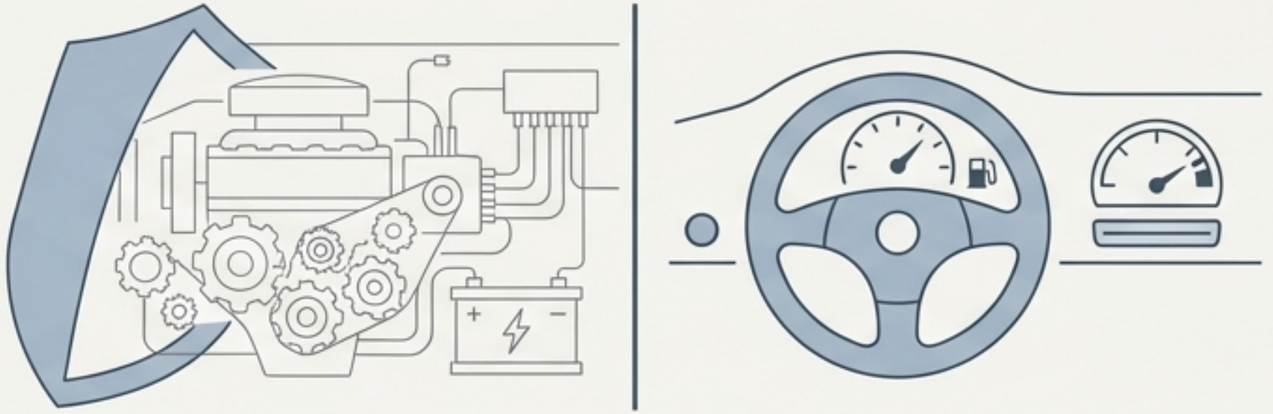
The `self` parameter is a reference to the current instance of the class. It is used to access the attributes and methods of the object itself (e.g., `self.balance`).

Usage Example

```
# Calling methods on an object
my_account = BankAccount("Charlie", 200)
my_account.deposit(50)  # Output: Deposited 50. New balance: 250
```

Pillar 1: Encapsulation - The Protective Shell

Hidden Implementation



Public Interface

Encapsulation protects an object's internal state from outside interference. It hides complex implementation details behind a clean public interface, making your code safer and easier to maintain.

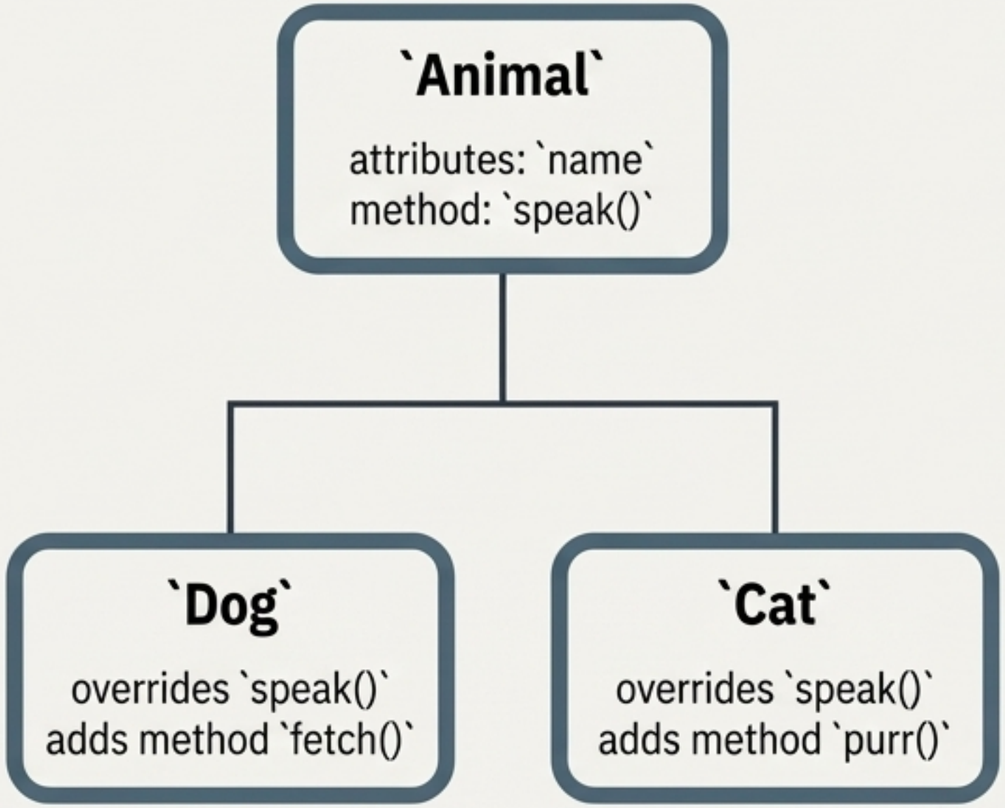
```
class User:
    def __init__(self, username, password):
        self.username = username
        self.__password = password # __ denotes a private attribute

    def check_password(self, attempt):
        # Public method to safely interact with private data
        return self.__password == attempt

# Using the class
user = User("admin", "s3cr3t_p@ss")
# print(user.__password) -> This would raise an AttributeError!
# print(user.check_password("wrong_pass")) -> False
```

Private Attributes

In Python, the __double_underscore prefix triggers 'name mangling'. This makes it harder for code outside the class to accidentally access or modify the attribute, effectively creating a private variable.

Pillar 2: Inheritance - Building on Existing Work

Inheritance allows a new class (child/derived) to take on the attributes and methods of an existing class (parent/base). This promotes code reuse and creates a logical hierarchy.



Parent Class Code Block

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        raise NotImplementedError(
"Subclass must implement")
```

Child Class Code Block

```
class Dog(Animal): # Dog "is-a" Animal
    def speak(self): # Overriding
the parent method
        return f"{self.name} says
Woof!"
```

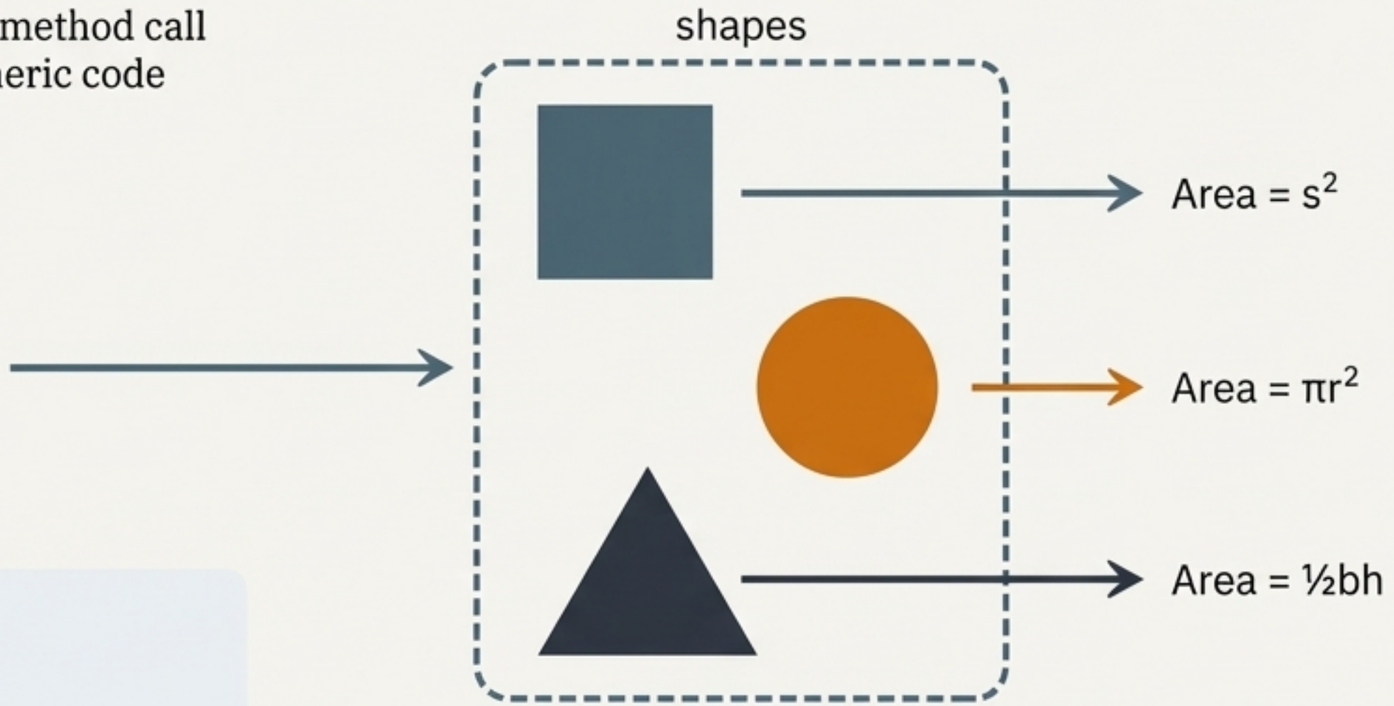
Key Terms

- **Parent/Base Class**: The class being inherited from (Animal).
- **Child/Derived Class**: The class that inherits (Dog).
- **Method Overriding:** Providing a specific implementation in the child class for a method already defined in the parent.
- **`super()`**: A function to call methods from the parent class.

Pillar 3: Polymorphism - One Interface, Many Forms

From the Greek for "many shapes," polymorphism is the ability of different objects to respond to the same method call in different ways. It allows for writing flexible, generic code that can work with objects of various types.

```
for shape in shapes:
    shape.calculate_area()
```




```python
class Shape:
    def area(self): raise NotImplementedError

class Rectangle(Shape):
    def __init__(self, w, h): self.w, self.h = w, h
    def area(self): return self.w * self.h

class Circle(Shape):
    def __init__(self, r): self.r = r
    def area(self): return 3.14 * self.r ** 2

# Polymorphism in action
shapes = [Rectangle(5, 10), Circle(7)]
for shape in shapes:
    # We don't need to know the specific type of shape,
    # just that it has an .area() method.
    print(f"Area: {shape.area()}")
```

Duck Typing in Python

If it walks like a duck and quacks like a duck, it's a duck. We care about what an object *can do* (its methods), not what it *is** (its class).

Integrating with Python: Special (Dunder) Methods

Make your objects feel native.

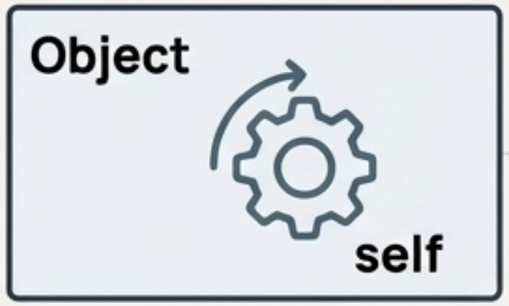
Python has a set of special methods, often called “magic” or “dunder” (double underscore) methods, that you can define to give your objects built-in behavior. This is called operator overloading.

Method	Operator/Function	Description
`__init__(self, ...)`	`MyClass()`	Object initialization (constructor)
`__str__(self)`	`str()`, `print()`	User-friendly string representation
`__repr__(self)`	`repr()`	Developer-friendly representation
`__len__(self)`	`len()`	Get the length of the object
`__add__(self, other)`	`+`	Defines addition
`__eq__(self, other)`	`==`	Defines equality comparison

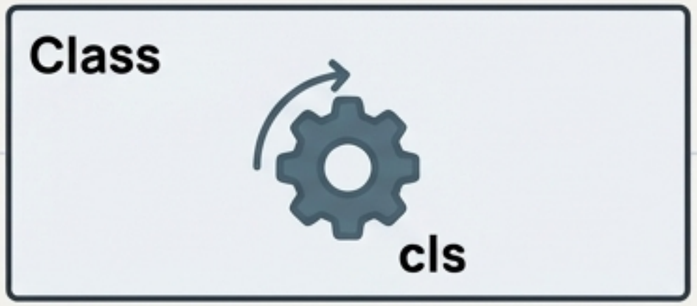
Example: A Custom Vector Class

```
class Vector:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(2, 3)
v2 = Vector(5, 1)
v3 = v1 + v2  # This works because of __add__!

print(v3) # Output: Vector(7, 4)
```

Advanced Design: Class and Static Methods



Instance Method: Operates on an instance's state.



Class Method: Operates on the class; can be used as an alternative constructor.

Static Method: A utility function related to the class, but doesn't need instance or class state.

A `Date` Class with a Factory Method

```
class Date:
    def __init__(self, day, month, year):
        self.day, self.month, self.year = day, month, year

    @classmethod
    def from_string(cls, date_string): # A "factory" method
        # Parses "dd-mm-yyyy"
        day, month, year = map(int, date_string.split('-'))
        return cls(day, month, year) # Creates an instance

    @staticmethod
    def is_valid_date(date_string):
        # A utility function that doesn't need cls or self
        # ... validation logic ...
        return True
```

When to Use Which

- **Instance Method:** The default. Needs access to an object's data (`self`).
- **Class Method:** To create factory methods that produce instances of the class in alternative ways.
- **Static Method:** For utility functions that have a logical connection to the class but don't depend on class or instance state.

The Architect's Choice: Inheritance ('Is-A') vs. Composition ('Has-A')

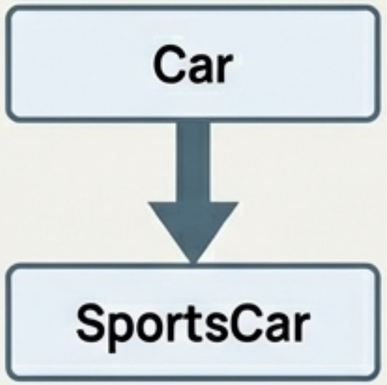
Inheritance (Is-A)

Establishes a hierarchical, parent-child relationship.

A `SportsCar` **is a** `Car`.

Pro: Reuses code directly from the parent.

Con: Creates tight coupling. Changes in the parent can break the child.



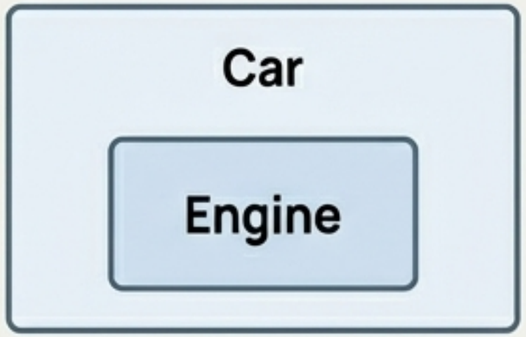
Composition (Has-A)

Builds complex objects by combining simpler ones.

A `Car` **has an** `Engine`.

Pro: More flexible and loosely coupled. You can swap out component parts.

Con: Can require more boilerplate code to wire components together.



Prefer composition over inheritance. It often leads to more flexible, scalable, and understandable designs. Start by asking if the relationship is 'is-a' or 'has-a'.

Capstone Project: Building a Library Management System
Part 1: Defining the Core Entities

Goal: To model a simple library system that manages a collection of books and a list of members who can borrow them, demonstrating all major OOP principles.

The `Book` Class

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self._isbn = isbn # Protected attribute
        self.is_checked_out = False

    def __repr__(self): # Dunder method for clear
    representation
        return f"'{self.title}' by {self.author}"
```

The `Member` Class

```
class Member:
    def __init__(self, name, member_id):
        self.name = name
        self.member_id = member_id
        self.borrowed_books = [] # A list of Book
        objects

    def __repr__(self):
        return f"Member: {self.name} (ID:
        {self.member_id})"
```

We begin by creating our 'blueprints'. The `Book` and `Member` classes encapsulate the data and state for the fundamental objects in our system.

Capstone Project: Building a Library Management System

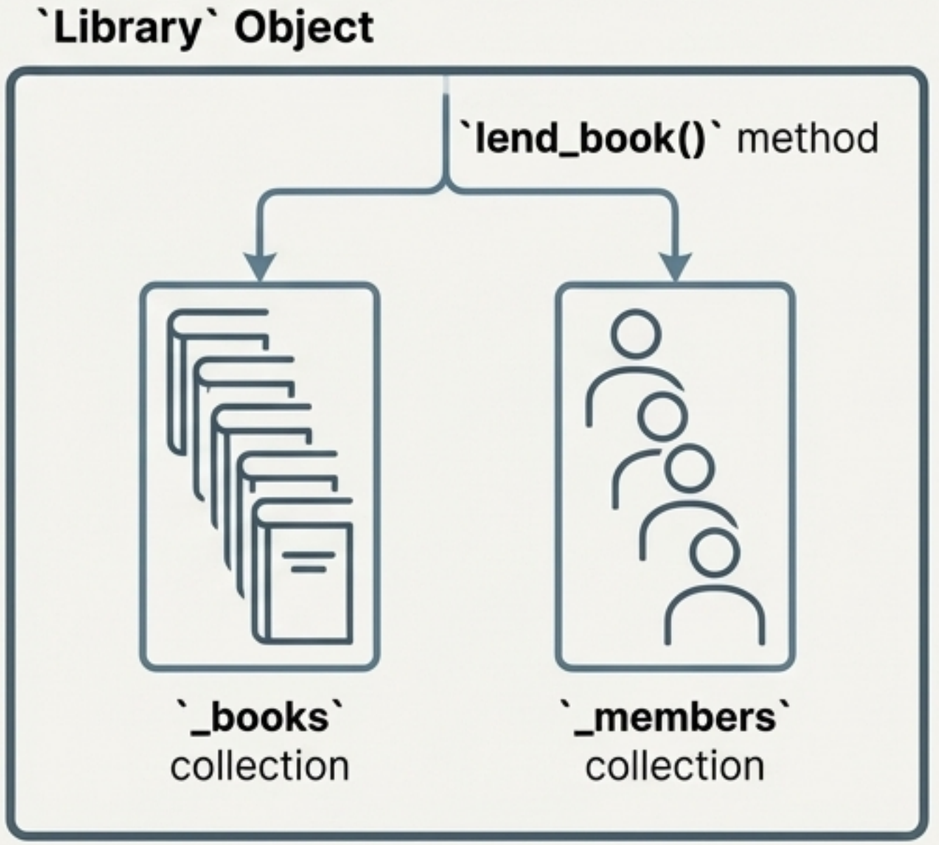
Part 2: The `Library` as an Orchestrator

Focus: Composition & Encapsulation

The `Library` class doesn't inherit from `Book` or `Member`. Instead, it *has* a collection of them. It acts as the central hub, providing a public interface to manage the system's core logic while hiding the implementation details.

The `Library` Class (Partial)

```
class Library:
    def __init__(self):
        self._books = {} # Using dict for fast lookup by ISBN
        self._members = {}

    def add_book(self, book):
        self._books[book._isbn] = book

    def add_member(self, member):
        self._members[member.member_id] = member

    def lend_book(self, isbn, member_id):
        book = self._books.get(isbn)
        member = self._members.get(member_id)

        if book and member and not book.is_checked_out:
            book.is_checked_out = True
            member.borrowed_books.append(book)
            print(f"Lent '{book.title}' to {member.name}")
        else:
            print("Lending failed.")
```



Capstone Project: Building a Library Management System

Part 3: Extending the System with Inheritance

Focus: Inheritance & Polymorphism

The true power of an OOP design is its extensibility. What if we want to add digital books with different lending rules? With inheritance, we can create a specialized `DigitalBook` class without changing our existing `Book` or `Library` code.

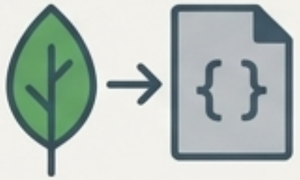
```
class DigitalBook(Book): # DigitalBook "is-a" Book
    def __init__(self, title, author, isbn, file_format):
        super().__init__(title, author, isbn) # Call parent constructor
        self.file_format = file_format

    # We could override lending logic here if needed,
    # but for now, it inherits the base behavior.
```

Polymorphism in Action

Our `Library` class's methods can work with `Book` objects and `DigitalBook` objects interchangeably, as long as they share the same interface. This is polymorphism.

```
library = Library()
physical_book = Book("The Hobbit", "J.R.R. Tolkien", "123")
digital_book = DigitalBook("The Silmarillion", "J.R.R. Tolkien", "456", "EPUB")

library.add_book(physical_book)
library.add_book(digital_book) # Works seamlessly!
```

The Object-Oriented Mindset

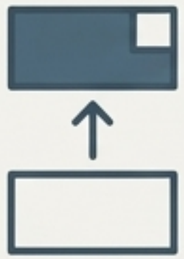


Model the Real World

Think in terms of objects, their properties, and their behaviors. Your code structure should reflect the problem domain.

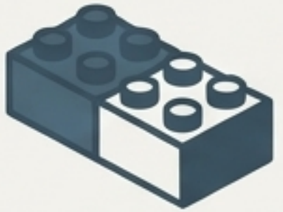
Bundle Data and Behavior (Encapsulation)

Keep related data and the functions that operate on it together in one place. Protect it from the outside world.



Build on What You Have (Inheritance)

Don't reinvent the wheel. Extend existing code to create specialized versions.



Write Flexible Code (Polymorphism & Composition)

Design components that can be swapped and interchanged. Depend on interfaces, not on concrete implementations.

Mastering these principles allows you to build systems that are not just functional, but also resilient, scalable, and a pleasure to maintain.