CHALLENGE / BONUS FEATURES:

- **Add a __str__ method** to your Car class to give it a nice string representation, like "A blue Toyota".
- **Add more attributes** in __init__, such as model and year. Update the __str__ method to include them.
- **Create a state-aware method:** Add a method drive(self, distance). This method should only work if self.is_running is True. If the car is off, it should print a message telling the user to start the car first. If it's on, it should print f"Driving {distance} miles.".

THE POWER OF INHERITANCE

- This core OOP principle allows a new class (child) to reuse and extend the functionality of an existing class (parent). Instead of duplicating code, you place the common attributes and methods in the parent and let child classes inherit them. For example, a Dog and a Cat both share behaviors like eat() and sleep() from an Animal class but can also have their own unique methods like bark() or meow(). This makes code more organized, reusable, and easier to maintain.

THE POWER OF POLYMORPHISM

- Polymorphism means “many forms” — the same method name can behave differently depending on the object. It allows us to write code that can handle different types of objects in a uniform way. For example, both a Circle and a Rectangle can have an area() method, but each calculates it differently. This makes code more flexible and scalable.

HANDS-ON LAB: THE SHAPE CLASS

- Goal: Define a Shape base class and create child classes like Circle and Rectangle so the same method name (area) behaves differently across types.
- Step-by-Step Instructions :
  - Create a new file shape_class.py

PART 1 — THE BLUEPRINT (__INIT__)

- Define a class named Shape.
- Inside Shape, define a method area() that contains only pass (placeholder).

PART 2 — ADDING BEHAVIORS (METHODS)

- Define a class Circle that inherits from Shape.
- Add an __init__ that accepts radius and sets self.radius.
- Implement area() in Circle to return 3.14 * self.radius * self.radius.
- Define a class Rectangle that inherits from Shape.
- Add an __init__ that accepts width and height and sets self.width, self.height.
- Implement area() in Rectangle to return self.width * self.height.

PART 3 — TESTING YOUR CLASSES

- Create a list shapes = [Circle(5), Rectangle(4, 6)].
- Loop over shapes and call shape.area() for each item.
- Observe that each object responds to area() in its own way.

CHALLENGE / BONUS FEATURES:

- Add a Triangle class with base and height; implement area() as 0.5 * base * height.
- Implement __str__ for each shape to return a friendly description.
- Write total_area(shapes) that sums area() for any list of Shape objects.

THE POWER OF ABSTRACTION

- Abstraction is about hiding unnecessary details and exposing only what's essential. In Python, this is often done with abstract classes that define a contract. Subclasses must implement the abstract methods, but each can do so in their own way. For example, a PaymentMethod class may define pay(), but how the payment happens depends on whether it's CreditCard, PayPal, or Cash. This makes systems more organized and extensible.

HANDS-ON LAB: THE PAYMENT SYSTEM

- Goal: Define a PaymentMethod abstract base class that specifies a contract, then implement concrete payment classes that follow it.
- Step-by-Step Instructions :
  - Create a new file payment_system.py

PART 1 — THE BLUEPRINT (ABSTRACT CLASS)

- Import ABC and abstractmethod from the abc module.
- Define a class PaymentMethod that inherits from ABC.
- Inside it, declare an abstract method pay(self, amount) using @abstractmethod.

PART 2 — ADDING BEHAVIORS (METHODS)

- Define a class CreditCard that inherits from PaymentMethod.
- Implement pay(self, amount) to print "Paid $<amount> using Credit Card".
- Define a class PayPal that inherits from PaymentMethod.
- Implement pay(self, amount) to print "Paid $<amount> using PayPal".

PART 3 — TESTING YOUR CLASSES

- Create a list methods = [CreditCard(), PayPal()].
- Loop for m in methods: m.pay(100) and observe each implementation run.
- Note that you cannot instantiate PaymentMethod directly — it enforces the contract.

CHALLENGE / BONUS FEATURES:

- Create a Cash class that implements pay() (no external API required).
- Add a new abstract method refund(self, amount) to PaymentMethod and implement it in all subclasses.
- Add simple validation (e.g., raise ValueError for negative amounts).
- Extend to support currencies (add a currency attribute and format outputs).

CLASS ATTRIBUTES VS INSTANCE ATTRIBUTES

- So far, we've only seen instance attributes — data unique to each object. But sometimes, we need information that is shared by all objects of a class. These are called class attributes.
- Instance attributes → unique per object (e.g., each student has their own name, age, ID).Class attributes → shared across all objects (e.g., all students belong to the same school).This distinction helps us track data at both the individual and collective level.

CODE EXAMPLE:

- class Dog:
- species = "Canis familiaris"
- count = 0
- def __init__(self, name):
- self.name = name
- Dog.count += 1

- print(Dog("Buddy").name, Dog.count) # Buddy 1

MODIFYING & DELETING OBJECT PROPERTIES

- Python objects are dynamic. After creating an object, you can:
- Modify existing attributes
- Add new attributes
- Delete attributes
- This flexibility makes Python objects behave like “living” entities that can evolve over time.

CODE EXAMPLE:

- class Car:
-     def __init__(self): self.color="blue"
- c = Car(); c.color="red"   # modify
- c.mileage=50000   # add
- del c.mileage   # delete
- print(getattr(c,"mileage","N/A")) # N/A

PYTHON NAMING CONVENTIONS & PRIVACY

- Python does not enforce access control like private/protected in Java or C++. Instead, it uses naming conventions to signal intent:
- Public (default):
  - Accessible by everyone.
- Protected (_single_underscore):
  - Internal use only, but still accessible.
- Private (__double_underscore):
  - Python applies name mangling to discourage direct access.This helps developers respect boundaries without restricting flexibility.

CODE EXAMPLE:

- class Account:
-     def __init__(self):
-         self.owner="Alice"   # public
-         self._bal=1000   # protected
-         self.__pin=1234   # private

- a=Account()
- print(a.owner, a._bal, a._Account__pin)

HANDS-ON LAB: THE ANIMAL CLASS

- **Goal:** To apply all the core OOP concepts—Encapsulation, Inheritance, Polymorphism, and Abstraction—by creating a system of related classes. This lab will also reinforce the use of class attributes and naming conventions.
- **File:** Create a new file named zoo.py.

PART 1: THE ABSTRACT BLUEPRINT (ABSTRACTION)

- **Concept:** We'll start by defining a "contract." We know every animal in our zoo needs to eat and make a sound, but we don't know *how* yet. This is a perfect use case for an abstract base class.
- **Instructions:**
  - Import ABC and abstractmethod from the abc module.
  - Define a class named Animal that inherits from ABC.
  - In the __init__ method, it should accept a name and species. Create self.name and self.species as **instance attributes**.
  - Define a class attribute zoo_name = "The Python Zoo". Class attributes are shared by all instances.
  - Declare an abstract method make_sound(self) using the @abstractmethod decorator. It should only contain pass.
  - Declare another abstract method eat(self). It should also only contain pass.

PART 2: CREATING CONCRETE CLASSES (INHERITANCE)

- **Concept:** Now we'll create specific types of animals. These "child" classes will inherit the common attributes (name, species, zoo_name) from the Animal "parent" class and provide their own concrete implementations for the abstract methods.
- **Instructions:**
- **Create a Lion class that inherits from Animal.**
  - In its __init__, it should call the parent's __init__ method using super().__init__(name, species="Lion").
  - Implement the make_sound() method to print "Roar!".
- **Create a Monkey class that inherits from Animal.**
  - In its __init__, use super().__init__(name, species="Monkey").
  - Implement make_sound() to print "Ooh ooh aah aah!".
  - Implement eat() to print f"{self.name} peels and eats a banana.".
- **Create a Zookeeper class (this one does *not* inherit from Animal).**
  - Its __init__ should take a name.
  - Add a **protected** instance attribute self._employee_id and set it to a random number (e.g., random.randint(1000, 9999)—don't forget to import random).
  - Add a **private** instance attribute self.__salary and set it to a value like 45000.

PART 3: BRINGING THE ZOO TO LIFE (POLYMORPHISM & CLASS ATTRIBUTES)

- Concept: The power of polymorphism is that we can treat different objects in a uniform way. We can ask any Animal to make_sound(), and it will respond correctly based on its specific class. We will also see how class attributes work.
- Instructions:
- Create a list called animals and populate it with instances of your concrete classes:
- code

```
Python
animals = [
    Lion("Leo"),
    Monkey("Momo")
]
```

- Create an instance of your Zookeeper class: zookeeper = Zookeeper("Hamza").
- Demonstrate Polymorphism:
- Write a for loop that iterates through the animals list.
- Inside the loop, call animal.make_sound() and animal.eat() for each animal.
- Observe how each object responds to the same method call in its own unique way.
- Demonstrate Class Attributes:
- For each animal in your loop, print f"{animal.name} lives at {animal.zoo_name}".
- Notice that you can access zoo_name from both the instance (leo.zoo_name) and the class itself (Lion.zoo_name).

CHALLENGE / BONUS FEATURES (APPLYING ALL CONCEPTS)

- **Modify an Object:** After creating your Lion object, add a new attribute to it directly: leo.favorite_toy = "Big Red Ball". Print this new attribute to show that Python objects are dynamic.
- **Naming Conventions & Privacy:**
  - Try to print the zookeeper's protected ID: print(zookeeper._employee_id). Observe that it works, but the underscore signals that you *shouldn't* do this.
  - Try to print the zookeeper's private salary: print(zookeeper.__salary). Observe that this causes an AttributeError.
  - Now, access the "mangled" name: print(zookeeper._Zookeeper__salary). This demonstrates how name mangling works to discourage direct access.

- **Extend with a New Class:**
  - Create a Penguin class that inherits from Animal.
  - Implement its make_sound() and eat() methods.
  - Add a new method unique to penguins, like slide(), that prints f"{self.name} slides on its belly!".
  - Add a penguin to your animals list and re-run your loop.
- **Create a feed_animals Method:**
  - Add a method to your Zookeeper class called feed_animals(self, animals_list).
  - This method should take a list of Animal objects as a parameter.
  - It should loop through the list and call the eat() method on each animal, demonstrating how different classes can interact. Call this method with your list of animals.