## DAY 8: EVENING ASSESSMENT 1. Employee Salary with Bonus Logic

class Employee:  
 def \_\_init\_\_(self, name, base\_salary):  
 self.\_\_name = name  
 self.\_\_salary = base\_salary  
  
 def add\_bonus(self, bonus):  
 if bonus < 0:  
 raise ValueError("Bonus cannot be negative.")  
 self.\_\_salary += bonus  
  
 def get\_details(self):  
 return f"Employee: {self.\_\_name}, Salary: {self.\_\_salary}"  
  
emp = Employee("Alice", 50000)  
emp.add\_bonus(5000)  
print(emp.get\_details())

EXPLANATION:

* Private Variables:\_\_name and \_\_salary are private. They can't be accessed directly from outside the class.
* Constructor (\_\_init\_\_): Runs automatically when an object is created. Sets the employee’s name and starting salary.
* Bonus Check: The add\_bonus() method checks if the bonus is negative.
* Formatted Output: get\_details() gives a clean and readable message using f-strings.
* Object-Oriented: The code uses class and objects

## 2. Validated Bank Account with Deposit and Withdraw

class BankAccount:  
 def \_\_init\_\_(self, owner, balance):  
 self.\_\_owner = owner  
 self.\_\_balance = balance  
  
 def deposit(self, amount):  
 if amount <= 0:  
 raise ValueError("Invalid deposit amount.")  
 self.\_\_balance += amount  
  
 def withdraw(self, amount):  
 if amount > self.\_\_balance:  
 raise ValueError("Insufficient funds.")  
 self.\_\_balance -= amount  
  
 def get\_balance(self):  
 return self.\_\_balance  
  
acc = BankAccount("John", 1000)  
acc.deposit(500)  
acc.withdraw(200)  
print("Balance:", acc.get\_balance())

EXPLAINATION:

* Private Variables: \_\_owner and \_\_balance are private
* acc = BankAccount("John", 1000) creates an object of the class.
* Access is controlled through methods like deposit(), withdraw(), and get\_balance().
* Internal logic like how deposit/withdraw work is hidden from the user. User just calls methods like acc.deposit(500) without knowing internal calculations.
* Trying to access acc.\_\_balance directly will give an error
* The BankAccount class can be reused to create multiple accounts like acc1, acc2, etc., each with their own data.

## 3. Encapsulation with Password Protection

class User:  
 def \_\_init\_\_(self, username, password):  
 self.\_\_username = username  
 self.\_\_password = password  
  
 def authenticate(self, input\_password):  
 return self.\_\_password == input\_password  
  
 def get\_username(self):  
 return self.\_\_username  
  
user = User("admin", "12345")  
print(user.authenticate("12345"))  
print(user.authenticate("abc"))

EXPLANATION:

* Direct access like user.\_\_password will give an error.
* user = User("admin", "12345") creates a user object with given credentials.
* User only needs to call authenticate("password"). The actual comparison logic is hidden inside the method.
* \_\_username and \_\_password are private. They can’t be accessed directly from outside the class — ensures data protection.
* The User class can be reused to create multiple users with their own credentials.

## 4. Encapsulated Stock Portfolio Tracker

class StockPortfolio:  
 def \_\_init\_\_(self):  
 self.\_\_stocks = {}  
  
 def add\_stock(self, symbol, quantity):  
 if quantity <= 0:  
 raise ValueError("Invalid quantity.")  
 self.\_\_stocks[symbol] = self.\_\_stocks.get(symbol, 0) + quantity  
  
 def remove\_stock(self, symbol, quantity):  
 if symbol not in self.\_\_stocks or self.\_\_stocks[symbol] < quantity:  
 raise ValueError("Not enough stock to remove.")  
 self.\_\_stocks[symbol] -= quantity  
  
 def get\_holdings(self):  
 return self.\_\_stocks  
  
portfolio = StockPortfolio()  
portfolio.add\_stock("AAPL", 10)  
portfolio.add\_stock("TSLA", 5)  
portfolio.remove\_stock("AAPL", 5)  
print(portfolio.get\_holdings())

EXPLANATION:

* Uses .get(symbol, 0) to safely update stock quantities. Ensures that if a stock doesn't exist yet, it starts from 0.
* Constructor (\_\_init\_\_): Runs automatically when StockPortfolio() is called. Initializes an empty dictionary to store stock symbols and their quantities.
* Object Creation: portfolio = StockPortfolio() creates a new object
* Internal logic like how stocks are stored is hidden. The user doesn’t need to know about the dictionary
* add\_stock() ➝ Adds new stock or increases quantity.
* remove\_stock() ➝ Decreases quantity, only if enough exists.
* get\_holdings() ➝ Returns the current stock holdings.

## 5. Student Grades with Private Data

class Student:  
 def \_\_init\_\_(self, name):  
 self.\_\_name = name  
 self.\_\_grades = []  
  
 def add\_grade(self, grade):  
 if not (0 <= grade <= 100):  
 raise ValueError("Invalid grade.")  
 self.\_\_grades.append(grade)  
  
 def get\_average(self):  
 return sum(self.\_\_grades) / len(self.\_\_grades)  
  
student = Student("Emma")  
student.add\_grade(90)  
student.add\_grade(80)  
print(f"Average: {student.get\_average()}")

EXPLANATION:

* Encapsulation: Grades are stored privately inside the class. You can only add or view them using methods
* Constructor (\_\_init\_\_): Automatically called when you create a Student object. Sets the student’s name and starts with an empty list of grades.
* add\_grade() checks that the grade is between 0 and 100. Prevents invalid values from being added.
* add\_grade() ➝ Adds a grade if it’s valid.
* get\_average() ➝ Calculates and returns the average of all grades.
* student = Student("Emma") creates a new student named Emma.
* \_\_name and \_\_grades are private.

## 6. Property Access with Read/Write Control

class Temperature:  
 def \_\_init\_\_(self):  
 self.\_\_celsius = 0  
  
 @property  
 def celsius(self):  
 return self.\_\_celsius  
  
 @celsius.setter  
 def celsius(self, value):  
 if value < -273.15:  
 raise ValueError("Invalid temperature.")  
 self.\_\_celsius = value  
  
temp = Temperature()  
temp.celsius = 25  
print(temp.celsius)

EXPLANATION:

* @property allows access to \_\_celsius like a normal variable even though it's private. This keeps the variable protected while still accessible in a clean way
* Internal logic is hidden from the user.
* User just writes temp.celsius = 25 without worrying about internal checks.
* temp = Temperature() creates an object.
* temp.celsius = 25 sets the value using the setter.
* print(temp.celsius) gets the value using the getter.
* @celsius.setter is used to control how the value is set.

## 7. Smart Lock Device

class SmartLock:  
 def \_\_init\_\_(self, pin):  
 self.\_\_pin = pin  
 self.\_\_locked = True  
  
 def unlock(self, input\_pin):  
 if input\_pin == self.\_\_pin:  
 self.\_\_locked = False  
 else:  
 print("Incorrect PIN")  
  
 def lock(self):  
 self.\_\_locked = True  
  
 def is\_locked(self):  
 return self.\_\_locked  
  
lock = SmartLock("1234")  
lock.unlock("1234")  
print("Locked?", lock.is\_locked())

EXPLANATION:

* \_\_pin and \_\_locked are private, protecting the lock status and PIN from direct access.
* The unlock() method checks the input PIN before unlocking, enforcing secure authentication.
* Users interact with the lock only through public methods like unlock(), lock(), and is\_locked().
* is\_locked() returns True/False, making it easy to check lock status programmatically.
* The user can lock and unlock the object without knowing how the internal data is stored or validated.

## 8. Employee Details with Computed Property

class Employee:  
 def \_\_init\_\_(self, name, salary):  
 self.\_\_name = name  
 self.\_\_salary = salary  
  
 @property  
 def annual\_salary(self):  
 return self.\_\_salary \* 12  
  
 def get\_name(self):  
 return self.\_\_name  
  
emp = Employee("Sara", 5000)  
print(emp.get\_name(), emp.annual\_salary)

EXPLANATION:

* \_\_name and \_\_salary are private, ensuring data is hidden and secure.
* annual\_salary is defined using @property, allowing access like a variable while hiding the calculation logic.
* Abstraction: The employee’s annual salary is calculated behind the scenes -user just accesses emp.annual\_salary.
* Encapsulation: get\_name() is used to retrieve the employee's name, maintaining controlled access.
* Combines data (name, salary) and behavior (get\_name, annual\_salary) inside a reusable class.

## 9. Encapsulated Voting System

class VotingMachine:  
 def \_\_init\_\_(self):  
 self.\_\_votes = {}  
  
 def vote(self, candidate):  
 self.\_\_votes[candidate] = self.\_\_votes.get(candidate, 0) + 1  
  
 def result(self):  
 return sorted(self.\_\_votes.items(), key=lambda x: x[1], reverse=True)  
  
vm = VotingMachine()  
vm.vote("Alice")  
vm.vote("Bob")  
vm.vote("Alice")  
print(vm.result())

EXPLANATION:

* Votes are stored in a dictionary (\_\_votes) where the key is the candidate’s name and the value is the number of votes.
* The vote() method adds a vote for the given candidate. If the candidate is not in the dictionary, it adds them with 1 vote.
* The result() method shows all candidates and their votes, sorted from highest to lowest.
* Encapsulation is used — the \_\_votes dictionary is private and can’t be accessed directly from outside.
* The program uses lambda with sorted() to sort the dictionary items by number of votes in descending order.

## 10. Hotel Room Booking with Access Control

class HotelRoom:  
 def \_\_init\_\_(self, room\_no):  
 self.\_\_room\_no = room\_no  
 self.\_\_is\_booked = False  
  
 def book(self):  
 if self.\_\_is\_booked:  
 raise Exception("Room already booked.")  
 self.\_\_is\_booked = True  
  
 def status(self):  
 return "Booked" if self.\_\_is\_booked else "Available"  
  
room = HotelRoom(101)  
room.book()  
print(room.status())

EXPLANATION:

* Each hotel room has a room number and a booking status, which starts as False.
* The book() method changes the room status to booked, but only if it’s not already booked.
* If you try to book a room that’s already booked, the program raises an exception saying “Room already booked.”
* The status() method checks and returns whether the room is “Available” or “Booked”.
* Encapsulation is used - the room number and booking status are private, and can only be accessed through methods.

## 11. Payment Interface using Abstraction

from abc import ABC, abstractmethod  
  
class Payment(ABC):  
 @abstractmethod  
 def pay(self, amount): pass  
  
class CreditCard(Payment):  
 def pay(self, amount):  
 print(f"Paid ₹{amount} using Credit Card")  
  
class UPI(Payment):  
 def pay(self, amount):  
 print(f"Paid ₹{amount} using UPI")  
  
def checkout(method: Payment, amt):  
 method.pay(amt)  
  
checkout(CreditCard(), 500)  
checkout(UPI(), 200)

EXPLANATION:

* Payment is an abstract base class, which means it can’t be used directly- it only defines a structure for payment types.
* The pay() method is abstract, so any class that inherits from Payment must implement it.
* The CreditCard and UPI classes override the pay() method to give their own payment messages.
* The checkout() function accepts any payment method that is a subclass of Payment, making it reusable and flexible.
* Polymorphism is used — the same checkout() function behaves differently depending on whether it's passed a CreditCard or UPI object.

## 12. Abstract Shape Class

from abc import ABC, abstractmethod  
  
class Shape(ABC):  
 @abstractmethod  
 def area(self): pass  
  
class Circle(Shape):  
 def \_\_init\_\_(self, radius):  
 self.radius = radius  
  
 def area(self):  
 return 3.14 \* self.radius \* self.radius  
  
sh = Circle(3)  
print("Area:", sh.area())

EXPLANATION:

* Shape is an abstract class that defines a common structure for all shapes using the abstract method area().
* Abstract methods don’t have a body - they must be implemented in any subclass that inherits from Shape.
* Circle is a subclass of Shape and provides its own definition of the area() method using the formula for a circle.
* The constructor in Circle initializes the radius, and this value is used in the area calculation.
* When we create a Circle object and call area(), it returns the correct area using the radius provided

## 13. Abstract Animal Sound Generator

from abc import ABC, abstractmethod  
  
class Animal(ABC):  
 @abstractmethod  
 def sound(self): pass  
  
class Dog(Animal):  
 def sound(self):  
 print("Woof")  
  
class Cat(Animal):  
 def sound(self):  
 print("Meow")  
  
animals = [Dog(), Cat()]  
for animal in animals:  
 animal.sound()

EXPLANATION:

* Animal is an abstract class with an abstract method sound(), which means all subclasses must define how the animal sounds.
* The Dog and Cat classes inherit from Animal and implement the sound() method with their own behavior.
* The abstract method acts like a rule, forcing every animal type to have a sound() method.
* A list of animal objects is created, and a loop calls the sound() method for each - showing polymorphism in action.
* Each object responds differently to the same method call, showing how abstract classes support flexible and reusable code.

## 14. Report Generator Template

from abc import ABC, abstractmethod  
  
class ReportGenerator(ABC):  
 def generate(self):  
 self.fetch\_data()  
 self.format\_data()  
 self.export()  
  
 @abstractmethod  
 def fetch\_data(self): pass  
  
 @abstractmethod  
 def format\_data(self): pass  
  
 def export(self):  
 print("Exporting as PDF")  
  
class SalesReport(ReportGenerator):  
 def fetch\_data(self):  
 print("Fetching sales data")  
  
 def format\_data(self):  
 print("Formatting data")

EXPLANATION:

* ReportGenerator is an abstract class that defines a template method generate() to organize the steps of report creation.
* it contains two abstract methods - fetch\_data() and format\_data() -which must be implemented by subclasses like SalesReport.
* The export() method is already defined in the abstract class and is reused as-is by the subclass to print “Exporting as PDF”.
* SalesReport is a concrete subclass that implements the fetch\_data() and format\_data() methods with specific logic for sales reports.
* This program demonstrates the Template Method Pattern, where the base class defines the overall process and subclasses fill in the details.

## 15. Abstract Logger with Subclasses

from abc import ABC, abstractmethod  
  
class Logger(ABC):  
 @abstractmethod  
 def log(self, message): pass  
  
class ConsoleLogger(Logger):  
 def log(self, message):  
 print("Console:", message)  
  
class FileLogger(Logger):  
 def log(self, message):  
 print("Writing to file:", message)  
  
logger = ConsoleLogger()  
logger.log("App started")

EXPLANATION:

* Logger is an abstract class that defines a general rule - all loggers must have a log() method.
* The log() method is abstract, so each subclass like ConsoleLogger and FileLogger must provide its own version.
* ConsoleLogger prints the log message to the console, showing how it handles logging in one way.
* FileLogger simulates writing logs to a file, showing another way to log messages, using the same method name.
* This program uses polymorphism, where the same method , behaves differently based on the class that implements it.

## 16. Interface for Machine Operations

from abc import ABC, abstractmethod  
  
class Machine(ABC):  
 @abstractmethod  
 def start(self): pass  
  
 @abstractmethod  
 def stop(self): pass  
  
class Fan(Machine):  
 def start(self):  
 print("Fan started")  
  
 def stop(self):  
 print("Fan stopped")  
  
fan = Fan()  
fan.start()  
fan.stop()

EXPLANATION:

* Machine is an abstract class that acts like an interface, defining two methods: start() and stop().
* These methods are abstract, so any class that inherits from Machine must implement both start() and stop().
* The Fan class implements the interface by providing its own version of start() and stop() methods.
* Creating a Fan object and calling its methods shows how the interface ensures consistent behavior across machines.
* This program demonstrates abstraction and interface implementation, making sure every machine has start and stop operations.

## 17. Plugin Architecture with ABC

from abc import ABC, abstractmethod  
  
class Plugin(ABC):  
 @abstractmethod  
 def execute(self): pass  
  
class SpellCheck(Plugin):  
 def execute(self):  
 print("Checking spelling")  
  
class GrammarCheck(Plugin):  
 def execute(self):  
 print("Checking grammar")  
  
for plugin in [SpellCheck(), GrammarCheck()]:  
 plugin.execute()

EXPLANATION:

* Plugin is an abstract base class, which sets a common rule that all plugins must have an execute() method.
* Each subclass like SpellCheck and GrammarCheck implements the execute() method, providing its own functionality.
* The execute() method is called the same way on each object, but the behavior changes depending on the plugin
* Using a loop to execute multiple plugins makes it easy to add or run more plugins in the future.
* This design shows how abstract classes can be used to build a flexible plugin system, where each plugin follows a shared interface.

## 18. Shape Drawing App

from abc import ABC, abstractmethod  
  
class Drawable(ABC):  
 @abstractmethod  
 def draw(self): pass  
  
class Rectangle(Drawable):  
 def draw(self):  
 print("Drawing rectangle")  
  
class Triangle(Drawable):  
 def draw(self):  
 print("Drawing triangle")  
  
def render(d: Drawable):  
 d.draw()  
  
render(Rectangle())  
render(Triangle())

EXPLANATION:

* Drawable is an abstract class that defines a common interface for all drawable shapes using the draw() method.
* Rectangle and Triangle classes implement the draw() method, providing specific behavior for each shape.
* The render() function accepts any object of type Drawable, making it reusable for any shape that follows the interface.
* Polymorphism is used -the same render() function works with different shape objects and calls the correct draw() method.
* This program shows how abstract classes support flexibility, allowing new shapes to be added easily by just implementing the draw() method.

## 19. Music Player with Interface

from abc import ABC, abstractmethod  
  
class MediaPlayer(ABC):  
 @abstractmethod  
 def play(self): pass  
  
class Mp3Player(MediaPlayer):  
 def play(self):  
 print("Playing MP3")  
  
class WavPlayer(MediaPlayer):  
 def play(self):  
 print("Playing WAV")  
  
Mp3Player().play()  
WavPlayer().play()

EXPLANATION:

* MediaPlayer is an abstract class, which acts like an interface that requires a play() method in all its subclasses.
* Mp3Player and WavPlayer implement the play() method, each providing different behavior based on the file type.
* The play() method is mandatory, because it is marked as an abstract method in the base class.
* This program shows polymorphism, as both players use the same method name (play()), but produce different outputs.
* The design is flexible and extendable, allowing more players like AacPlayer or FlacPlayer to be added easily by just implementing play().

## 20. Data Storage Abstraction

from abc import ABC, abstractmethod  
  
class Storage(ABC):  
 @abstractmethod  
 def save(self, data): pass  
  
class Database(Storage):  
 def save(self, data):  
 print(f"Saving to DB: {data}")  
  
class FileSystem(Storage):  
 def save(self, data):  
 print(f"Saving to file: {data}")  
  
def store(storage: Storage, data):  
 storage.save(data)  
  
store(Database(), "Customer Data")  
store(FileSystem(), "Log Data")

EXPLANATION:

* Storage is an abstract class that defines a general method save() which all storage types must implement.
* Database and FileSystem are subclasses that provide their own way of saving data -one to a database, the other to a file.
* The store() function accepts any object that follows the Storage interface, showing flexibility and reusability.
* Polymorphism is used - the same store() function behaves differently based on the storage type passed to it.
* This program demonstrates abstraction -the user just calls store() without needing to know how the data is actually saved.