20 Advanced Python Programs: Encapsulation and Abstraction

## 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())

1. This program defines a class to represent an **employee** with a **name** and **base salary.**
2. The \_\_name and \_\_salary variables are **private** , meaning they can't be accessed directly from outside the class. This is **encapsulation**—hiding internal data.
3. The add\_bonus() method allows adding extra salary (bonus) safely, but only if it's a **positive value.**
4. If someone tries to add a **negative bonus**, the program throws an error to **prevent mistakes or misuse.**
5. The get\_details() method provides a clean way to get the employee’s info, like name and updated salary, without exposing internal variables.

## 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())

1. This class manages a **bank account** with the account holder's name and current balance, both kept **private** which is an accesses specifier.
2. The deposit() method ensures that only **positive amounts** which is above zero are added to the account.
3. The withdraw() method only works if there is **enough balance** .It throws an Insufficient Funds Exception.
4. This protects the account from accidental or fraudulent actions using **encapsulation** concept .
5. The get\_balance() method allows users to see the balance **safely** without modifying it directly.

## 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"))

1. The class models a **user account** with a username and password, both of which are **hidden** from outside access.
2. The authenticate() method checks if the entered password matches the stored one.
3. This protects the password using **encapsulation**, so even if someone accesses the object, they can’t read or change the password.
4. Only the get\_username() method is public and returns the username.
5. This structure supports **secure login systems** where passwords remain confidential and the data remains safe.

## 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())

1. StockPortfolio() class helps track your **investment portfolio** using a private dictionary of stocks and their quantities.
2. You can **add stocks** through the add\_stock() method, which ensures that only valid i.e is a positive amounts are accepted.
3. The remove\_stock() method checks if you have enough of that stock before reducing the quantity.
4. The internal dictionary \_\_stocks is **hidden** you can’t change it directly.
5. The get\_holdings() method lets you see your current investments safely.
6. We add the data or information like name and quantity by creating a object called portfolio

## 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()}")

1. This program defines a Student class that stores the name and list of grades, both kept **private.**
2. Grades can be added through the add\_grade() method, which accepts only values between 0 and 100.
3. This ensures the student data is **valid and safe** from accidental inputs like negative marks or giving wrong result.
4. The average grade is calculated using get\_average() you don’t access or calculate it manually.
5. All internal logic is hidden, providing **clean and controlled access** to student data.This hiding of data is known as abstraction.

## 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)

1. This class stores temperature in **Celsius**, using a private variable \_\_celsius.
2. It uses a special feature called @property, which allows **reading** the value like a normal variable.
3. It also uses a @celsius.setter that controls how the temperature can be **updated.**
4. If someone tries to set an impossible temperature (below -273.15°C, absolute zero), it raises an error.
5. This keeps the data **scientifically accurate** and secure, while still being easy to use.

## 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())

1. This models a **smart door lock** that opens only with the correct **PIN code.**
2. The lock starts in a **locked** state and can be unlocked using the unlock() method.
3. If the wrong PIN is entered, the door remains locked and gives an error message.
4. The current lock status is private, and can only be checked using the is\_locked() method.
5. This is a good example of **security and encapsulation** where sensitive state and logic are hidden from users.

## 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)

1. The Employee class includes a private salary and name.
2. It introduces a **computed property** called annual\_salary, which multiplies monthly salary by 12 which is computed in @property .
3. This property can be **read like a variable**, but it is actually calculated and an object emp is created to give the input to the name and salary.
4. The get\_name() method provides access to the employee’s name.
5. This is a clean and safe way to offer **derived information** without exposing raw data.

## 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())

1. This program creates a **voting machine** using a private dictionary to store votes per candidate.
2. The vote() method lets you vote for a candidate, adding to their count.
3. The result() method returns a **sorted list of candidates** based on who got the most votes.
4. It doesn’t allow direct changes to the vote data—only controlled updates are possible.
5. This ensures a  **fair voting system.**

## 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())

1. This class tracks whether a **hotel room** is available or booked, using a private variable.
2. The book() method changes the status to “booked” only if it’s not already reserved.
3. If someone tries to book an already booked room, it shows an error message or an exception as "Room already booked." .
4. The status() method gives the current status of the room: **Available or Booked.**
5. This kind of **controlled access and validation** is crucial in real-world booking systems.

## 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)

1. This program creates a **blueprint (interface)** for different payment methods like UPI or Credit Card.
2. The Payment class is an **abstract class** that is you can’t use it directly, but it tells other classes what method they must include just like pay().
3. CreditCard and UPI are like children that inherit the rules from Payment, and they define **how** the payment is actually done.
4. When we call the checkout() function, we don’t care **which method** is used—we just call pay(), and it works.
5. This is useful in real life where we want flexibility: like choosing between Google Pay or Credit Card without rewriting our checkout system

## 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())

1. This code models the idea of a **generic shape**, without saying exactly what shape it is.
2. The Shape class has an abstract method area(), which means every specific shape must explain **how to calculate** its own area.
3. The Circle class shows one example—it defines area() using the formula for circles (πr²).
4. This makes the code **flexible and extendable**: tomorrow you can create a Square class or Rectangle by following the same format.
5. It’s like designing a “Shape calculator” where each shape brings its own rule, but the way we ask for the area is always the same.

## 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()

1. This program defines a **general animal type** with the expectation that all animals must be able to make a sound.
2. The Animal class says: “Every animal must have a sound() method,” but it doesn’t define what the sound is.
3. The Dog and Cat classes implement this by saying dogs go “Woof” and cats go “Meow.”
4. In the loop, the program doesn’t care if it’s a dog or a cat—it just calls sound() on each one.
5. This kind of setup is useful when you want a **common interface** for many types but each one does things in its own way—like a zoo app managing different animals.

## 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")

1. This program builds a **template** for generating reports in steps: fetch data, format data, and export.
2. The base class ReportGenerator defines the structure (the steps) but doesn’t explain how to do each step.
3. The child class SalesReport fills in the blanks by giving details about where to get the data and how to format it.
4. The generate() method in the base class **controls the process,** so all reports will follow the same flow even if their content is different.
5. This is very useful in **business applications** where you need to generate many types of reports (sales, employee, finance) with different data but the same format.

## 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")

1. This code creates a logging system where messages can be recorded in different ways (console or file).
2. The Logger class is abstract,it tells us every type of logger must have a log() method.
3. ConsoleLogger shows the message on the screen, while FileLogger might write it to a file (for now it just prints a message).
4. This setup allows us to change the way logs are handled **without changing the rest of the application** we can plug in any logger we want.
5. This kind of abstraction is used in **large software systems** to make debugging and tracking easier across different environments (development vs production).

## 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()

1. This program creates a **generic interface for any machine,** like a fan or motor.
2. The Machine abstract class requires every machine to have two basic actions: start() and stop().
3. The Fan class implements this interface by defining what happens when it starts and stops (it prints messages).
4. This makes the code **standard and predictable** you know every machine class will have start/stop features.
5. It’s like building a remote that works with any machine—you don’t care **how** the machine runs, just that it responds to start/stop commands.

## 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()

1. This program shows how to build a **plugin system,** where you can add features like spell check or grammar check without changing the main code.
2. The Plugin abstract class forces every plugin to have an execute() method.
3. Subclasses like SpellCheck and GrammarCheck implement their version of execute().
4. You can loop through different plugins and run them the same way, even if they do different things.
5. This is like having a modular app (like MS Word) where each feature is a plugin you can enable/disable easily.

## 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())

1. This program defines a drawing system using the Drawable abstract class that requires each shape to have a draw() method.
2. The Rectangle and Triangle classes implement how each shape should be drawn (for now, just printing text).
3. The render() function accepts any drawable shape and calls draw() on it.
4. This means the program can **draw any shape**, even ones not yet created, as long as they follow the interface.
5. This is great for **drawing apps or design tools** that work with many shapes and keep the code clean and extendable.

## 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()

1. This code represents a music player system that supports multiple audio formats like MP3 and WAV.
2. The abstract class MediaPlayer says every format must have a play() method.
3. The Mp3Player and WavPlayer classes implement this by printing out what they are playing.
4. The main program just calls play() without worrying about which format is being used.
5. This is how **real media players** like VLC or Spotify work—they support many formats through a common interface.

## 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")

1. This program defines a flexible storage system that can save data in different places like a database or file.
2. The abstract class Storage ensures every storage type has a save() method.
3. Database and FileSystem classes implement how saving is done in their respective places.
4. The store() function can save data using **any storage type**, just by passing the right object.

5. This abstraction is used in **real-world apps** where you want to switch between saving locally or in the cloud without changing much code.