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())

## 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())

## 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"))

## 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())

## 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()}")

## 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)

## 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())

## 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)

## 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())

## 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())

## 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)

## 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())

## 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()

## 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")

## 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")

## 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()

## 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()

## 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())

## 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()

## 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")