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())

o/p:Employee:Alice ,Salary:55000

Explanation:

Private Attributes that is  Name and salary are encapsulated using \_\_.

Constructor(\_\_init\_\_):It  initializes name and base salary.

Bonus Validation:add\_bonus() raises an error for negative bonus.

Salary Update: Valid bonus gets added to salary.

Details :get\_details()  returns name and updated salary.

**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())

o/p:Balance:1300

Explanation:

Private Data which is Owner and balance are encapsulated.

Constructor \_\_init\_\_() used.

Deposit is used to reject non-positive deposit amounts,returns value error

Withdrawal is used to check for sufficient funds before deduction.

Balance Tracking is used which keeps balance updated after transactions.

get\_balance() provides controlled access to 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"))

o/p:

True

False

Explanation:

Private Credentials are Username and password are private.

Constructor \_\_init\_\_() used.

Authentication Method is added which validates input password against stored one.

Encapsulation is done,which ensures password can't be accessed directly.

User Identity is done using get\_username() returns username.

Provides Security,that is it prevents unauthorized access through encapsulation.

**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())

o/p:{‘AAPL’:5,’TLSA’ :5}

Explanation:

Encapsulation is done,where stocks stored privately in a dictionary.

Constructor \_\_init\_\_() used.

Add Stock is used to add or update quantity of a stock.

Remove Stock is used to check and reduce quantity of a stock.

Quantity feature prevents invalid/negative transactions.

Portfolio that is get\_holdings() reveals current stock positions.

**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()}")

o/p:Average:85.0

Explanation:

Private Grades List is declared where it stores grades privately.

Constructor \_\_init\_\_() used.

Grade method Only allows grades between 0–100.

Add Gradeis used to Append valid grades to the list along with student name

Average is used to computes average of all grades.

Name: Keeps student name privately encapsulated to grades.

**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)

o/p:25

Explanation:

Private Celsiusis used to encapsulate temperature value.

Constructor \_\_init\_\_() used.

Property Decorators are used to control access to celsius

Validation is checked whether temperature is correct.

It rejects temperatures below absolute zero.

Getter Method is used to retrieve temperature safely.

Setter Method performs validation and sets temperature cleanly.

**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())

o/p:Locked? False

Explanation:

Private PIN & Lock State function is to keep PIN and lock status hidden.

Unlock is used to compare input with correct PIN.

Constructor \_\_init\_\_() used.

Lock/Unlock are used to update lock state accordingly.

Status is checked that is is\_locked()  reveals current state.

Access: Encapsulation is done, boolean value returned

**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)

o/p:Sara 60000

Explanation:

Private Data that is Name and salary are hidden.

Constructor \_\_init\_\_() used.

Property for Salary that is annual\_salary computed dynamically, property decorator used

No Setter which means Read-only property for safety.

get\_name() gives name securely.

Output: Multiplies monthly to 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())

o/p:[(‘Alice’,2),(‘Bob’,1)]

Explanation:

Private Vote Storage is used to keep votes in hidden dictionary.

Constructor \_\_init\_\_() used.

Vote Count is used to increment count per candidate.

Result is stored,It returns sorted results by vote count.

Simple Interface is used, where there are just vote() and result() methods.

Encapsulation is done, It Prevents direct tampering with vote data.

**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())

o/p:Booked

Explanation:

Private Room State where booking status is hidden.

Constructor \_\_init\_\_() used.

Booking checks booking,Prevents double booking.

Status Method , it returns whether room is booked or not.

Constructor is used to initialize room number and availability.

Access(controlled) is given to user only.User can only book via method.

->We declare &import abc module for all polymorphism problems.

**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)

o/p:Paid ₹500 using Credit Card

    Paid ₹200 using UPI

Explanation:

Abstract Class is Payment which defines the pay() method as abstract.

Concrete Classes are CreditCard and UPI implement the pay() method differently.

Polymorphism is done where it checkout() accepts any Payment-based object.

Payment method: new payment methods can be added

Output: Display payment method and amount paid.

**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())

o/p:Area:28.26

Explanation:

1. Abstract Class is Shape Class which declares area() as an abstract method.
2. Concrete Class is Circlewhich implements area() using radius.
3. Constructor is used to accept radius and stores it.
4. Polymorphism is done where object sh gives  area() through abstract base.
5. Result: Prints calculated area of the circle.

**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()

o/p:Woof

    Meow

Explanation:

Abstract Class is Animal which declares abstract method sound().

Subclasses are Dog, Cat which provide unique implementations of sound().

Polymorphism: Iterates,(for loop used) through animal objects calling sound().

Code is Reusable where it is easily extendable(can add more animal types) to more animal types.

Output: Prints sounds for each animal type.

**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")

o/p:Fetching sales data

Formatting data

Exporting as PDF

Explanation:

Template Design Pattern is used to generate() defines a fixed process flow.

Abstract classes are fetch\_data() and format\_data() must be implemented.

Concrete Class is SalesReport which implements abstract steps.

export() is used to share data among or across all reports.

Output: Shows step-by-step report generation messages.

**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")

o/p:Console: App started

Explanation:

Abstract Class is Logger which declares log() as abstract.

Subclasses are ConsoleLogger and FileLogger which give custom outputs.

Interface-Based Design is Logger can be swapped easily.

ConsoleLogger is used to log a message to console.

Output: Logs the message with source tag.

**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()

o/p:Fan started

Fan stopped

Explanation:

Abstract Class is Machine which defines start() and stop() as abstract.

Concrete Class Fan is used to implement both methods.

Encapsulation is done where it hides operation logic behind an interface.

fan.start() and fan.stop() methods are called at last

Output: Prints fan operation status.

**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()

o/p:Checking spelling

    Checking grammar

Explanation:

Abstract Class is Plugi which declares execute() abstractly.

Plugins SpellCheck & GrammarCheck ,theseImplement execute() individually.

Design is Easy to add/remove features.

Loop is added where each plugin runs via execute() in loop(for loop used)

Output: Shows action performed by each plugin.

**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())

o/p:Drawing rectangle

    Drawing triangle

Explanation:

Abstract Class is Drawable , it requires to be implemented.

Concrete Shapes that is Rectangle and Triangle provide their drawing .

render(): render() works with any Drawable.

Interface gives drawing capability/feature.

Output: Displays drawing message based on shape.

**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()

o/p:Playing MP3

    Playing WAV

Explanation:

Abstract ClassMedia Player is used to declare play() abstractly.

Formats Mp3Player, WavPlayer both give or implement their own play() behavior.

Encapsulation is done.Where media format logic is  hidden in class.

Flexible, Can add more formats easily.

Output: Shows what format is  played.

**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")

o/p:Saving to DB:Customer Data

    Saving to file:Log Data

Explanation:

Abstract Storage Interface requires save(data)method.

Concrete data are:Database and FileSystem they implement save().

store(): Works with any Storage object and is reused

At runtime:Gives storage type at runtime.

Output: Shows where data is saved (DB-customer data or file-log data).