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())

Output:

Employee: Alice, 55000

1.Employee class is initialized with name and salary attributes.

2.There are two methods, one for adding bonus to the salary and the other is to get details.

3.while adding bonus, if the bonus is less than 0 then you will get a ValueError Exception.

4.Here, salary and bonus are private variables i.e., can be accessed only in this class.

5. So, created an object to access these attributes and methods from the class.

## 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())

Output:

Balance: 1300

1. Here, BankAccount class is initialized with owner and balance attributes.

2.There are three methods. One is to deposit the amount, second is to withdraw the amount and third is to get the balance.

[3.If](http://3.if) deposit amount is less than 0, then you may get ValueError. And if the withdraw amount is greater than the balance amount then you may get InsufficientFunds Error.

4.Owner and balance are the private attributes i.e., they can only be accessed in this class.

[5.So](http://5.so), to access these attributes and methods we created an object.

## 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"))

Output:

True

False

1.Here, class user is initialized with username and password.

2.There are 2 methods. One is to authenticate the password and the other is to get the username.

3.Here, username and password are the private variables i.e., can only be accessed in this class.

[4.](http://4.if) The given password should match with the original password.

5.created an object to access all these attributes and methods.

## 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())

Output:

{'AAPL': 5, 'TSLA': 5}

1.The \_\_stocks dictionary is private, meaning it **cannot be accessed directly** from outside the class, ensuring controlled data access.

2.add\_stock(symbol, quantity) adds a specified quantity to a stock symbol.

3.If the symbol already exists, it adds to the existing quantity.

4.It validates the quantity to ensure it's **greater than 0**.

5.This method **decreases the quantity** of a stock.

## 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()}")

Output:

Average: 85

1.Here, Student class is initialized with name and grades attributes.

2.There are two methods, one to add grade and the other is to get the average.

3. If grade is less than 0 or greater than 100, then ValueError Exception arises.

4.Here, name and grade are the private variables.

5.created an object to access all the attributes of the class.

## 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)

Output:

25

1.The variable \_\_celsius is made private using double underscores (\_\_), which prevents direct access from outside the class.

2.@property allows temp.celsius to **access the value** of \_\_celsius like an attribute, while still calling the getter method behind the scenes.

3.The setter checks if the assigned temperature is above absolute zero (−273.15°C). If not, it raises a ValueError.

4.When temp.celsius = 25 is executed, it calls the **setter method**, validating and storing the value.

5.When print(temp.celsius) is called, it triggers the **getter method** and prints 25.

## 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())

Output:

“Locked?” False

1.Here, Smartlook class is initialized with pin attribute.

2. Locked and pin are the private variables

3.given pin should match with the original pin to unlock the lock.

4.created an object to access all the attributes and methods from the class.

5.There are three methods, one is to check the match of given pin, second is for lock and third is to check the status of lock.

## 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)

Output:

Sara, 60000

1. Here, employee class is initialized with name and salary.

2. There are two methods, one is to calculate the annual salary from monthly salary and the other is to get the name.

3. Here, name and salary are tye private attributes.

[4.To](http://4.to) get the annual salary, one must multiply monthly salary with 12.

5.created an object to access all the attributes and methods from the 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())

Output:

Alice:2, Bob:1

1. Here, vothing machine class is initialized with dictionary called votes.

2. There are 2 methods, one for casting vote and the other is for result

3.Here, votes are private attributes.

4. As, the votes get casted, the values of the respective members in the dictionary is increased by one.

5.created an object to access all the attributes and methods of the class.

## 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())

Output:

Available

1. Hotelroom class is initialized with roomno.

2. There are two methods, one for booking and the other is to check the status.

3. Here,roomno. And is\_booked are private variables.

4. If the room is already booked, then an exception raises saying the room is already booked. If not, it shows available.

5.created an object to access all the attributes and methods of the class.

## 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)

Output:

Paid 500 using credit card

Paid 200 using UPI

1.Abstract method and ABC are imported from the abc library.

2.There is one abstract method and three concrete methods.

3. As, abstract method is present, this class is an abstract class.

4.Abstract method is defined with @abstractmethod.

5.Payment method is the abstract class with no implementation and credit card and upi are the concrete methods inheriting the payment method.

## 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())

Output:

Area:28.26

1.Here,shape class is the abstract class with abstract method area.

2. Shape class is inherited to Circle class which has concrete methods.

3. As area method is the abstract method in the shape class, it has no implementations.

4. It’s implementations are done in the circle class.

5. So, to access this area method in circle class, we created an object.

## 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()

Output:

Woof

Meow

1. Here, Animal class is an abstract class with an abstract method called sound

2. As it is an abstract class, there are no implementations.

3.This abstract class is further inherited to Dog class and Cat class for its implementations.

4.objects are created to access these classes.

5. These objects are stored in a list and iterated one by one.

## 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. Here, abstractmethod and ABC are imported from abc.

2. There are two abstract methods and one concrete method in the ReportGenerator class.

3.This ReportGenerator class is further inherited to salesreport class for implementations of the abstract methods from parent class.

4.Abstract methods are defined with @abstractmethod

5. As there are two abstract methods in the ReportGenerator class, this class becomes abstract class.

## 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")

Output:

Console: App Started

1.Here, ABC AND Abstract methods are imported from abc.

2.class Logger has an abstract method called log, so this class becomes an abstract class.

3. As abstract methods doesn’t have any implementations, these are inherited to other classes called ConsoleLogger and FileLogger

4. Implementations for the method log are done in these concrete methods.

5.crerated an object for these classes to access these attributes and methods.

## 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()

Output:

Fan Started

Fan Stopped

1.abstractmethod and ABC are imported from abc

2.class machine has two abstract methods called start and stop.

3.Since this class has abstract methods, this class becomes an abstract class.

4.These implementations are done in their extended classes.

5. This machine class is further inherited to fan class where implementations are done for these abstract methods.

## 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()

Output:

Checking Spelling

Checking grammar

1.abstractmethod and ABC are imported from abc.

2.Plugin is an abstract class with abstract method called execute.

3. This class is further inherited to the classes called Spellcheck and Grammarcheck.

4.These classes provide implementations for the abstract methods.

5.created objects for the accessing of these attributes and methods.

## 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())

Output:

Drawing Rectangle

Drawing Triangle

1.abstractmethod and ABC are imported from abc

2.Drawable is an abstract class with abstract method draw.

[3.As](http://3.as) abstract methods doesn’t have any implementations, this class is further inherited to the other classes.

4. There is a function called render which has created an object for the drawable class

5.Draw method is further implemented in these classes.

## 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()

Output:

Playing MP3

Playing Wav

1.abstractmethod and ABC are imported from abc.

2.Mediaplayer is an abstract class with abstract method called play.

3. This class is further inherited to the classes called MP3Player and Wavplayer

4.These classes provide implementations for the abstract methods.

5.created objects for the accessing of these attributes and methods.

## 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")

Output:

Saving to DB: Customer Data

Saving to file: Log Data

1. abstractmethod and ABC are imported from abc.

2.Storage is an abstract class with abstract method called save.

3. This class is further inherited to the classes called Database and FileSystem

4.These classes provide implementations for the abstract methods.

5.created objects for the accessing of these attributes and methods.