Experiment 10 - Encapsulation, Inheritance, and Polymorphism in Python

Name: Mohammad Sayeed

Roll no : C56 Div: C Class: TY CSE

1. BankAccount (private balance)

Section: Encapsulation

Problem Statement:

Create a class BankAccount where account_holder is public, _account_type is protected, and __balance is private. Write methods to deposit and withdraw money with validation.

Code:

```
class BankAccount:
    def __init__(self, holder, account_type, balance=0):
        self.account_holder = holder
        self._account_type = account_type
        self.__balance = balance
    def deposit(self, amt):
        if amt>0: self.__balance += amt
    def withdraw(self, amt):
        if 0<amt<=self.__balance: self.__balance -= amt
        else: print('Insufficient or invalid amount')
    def get_balance(self):
        return self.__balance

a=BankAccount('Rahul','Savings',1000)
a.deposit(500); a.withdraw(300)
print('Balance:', a.get_balance())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Balance: 1200
```

2. Student (marks private)

Section: Encapsulation

Problem Statement:

Design a class Student where name is public, _roll_no is protected, and __marks is private. Ensure marks can only be set between 0 and 100.

Code:

```
class StudentEnc:
    def __init__(self,name,roll):
        self.name=name; self._roll_no=roll; self.__marks=0
    def set_marks(self,m):
        if 0<=m<=100: self.__marks=m
        else: print('Invalid marks')
    def get_marks(self): return self.__marks

s=StudentEnc('Amina','R01'); s.set_marks(85); s.set_marks(150)
print('Marks:', s.get_marks())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Invalid marks
Marks: 85
```

3. Employee (salary private)

Section: Encapsulation

Problem Statement:

Write a class Employee with name as public, _department as protected, and __salary as private. Provide methods to set and get salary safely.

Code:

```
class EmployeeEnc:
    def __init__(self,name,dept):
        self.name=name; self._department=dept; self.__salary=0
    def set_salary(self,s):
        if s>=0: self.__salary=s
    def get_salary(self): return self.__salary

e=EmployeeEnc('Vikram','IT'); e.set_salary(50000); print('Salary:',
e.get_salary())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Salary: 50000
```

4. Book (copies private)

Section: Encapsulation

Problem Statement:

Create a class Book where title is public, _author is protected, and __copies is private. Make sure copies cannot go below zero.

Code:

```
class BookEnc:
    def __init__(self,title,author,copies):
        self.title=title; self._author=author; self.__copies=copies
    def add_copies(self,n): self.__copies+=n
    def remove_copies(self,n):
        if n<=self.__copies: self.__copies-=n
        else: print('Not enough copies')
    def copies(self): return self.__copies

b=BookEnc('Py','Auth',2); b.remove_copies(1); b.remove_copies(5);
print('Copies:', b.copies())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Not enough copies
Copies: 1
```

5. Car (speed private)

Section: Encapsulation

Problem Statement:

Develop a class Car with brand as public, _model as protected, and __speed as private. Speed should only change using accelerate and brake methods.

Code:

```
class CarEnc:
    def __init__(self,brand,model): self.brand=brand;
self._model=model; self.__speed=0
    def accelerate(self,inc): self.__speed += inc
    def brake(self,dec): self.__speed = max(0, self.__speed - dec)
    def speed(self): return self.__speed

c=CarEnc('Hyundai','i20'); c.accelerate(60); c.brake(20);
print('Speed:', c.speed())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Speed: 40
```

6. ATM (pin & balance private)

Section: Encapsulation

Problem Statement:

Make a class ATM with bank_name as public, _location as protected, and __pin and __balance as private. Allow deposit/withdraw only if pin matches.

Code:

```
class ATM:
    def __init__(self,bank,loc,pin,balance=0):
        self.bank_name=bank; self._location=loc; self.__pin=pin;
self.__balance=balance
    def deposit(self,pin,amt):
        if pin==self.__pin and amt>0: self.__balance+=amt
    def withdraw(self,pin,amt):
        if pin==self.__pin and 0<amt<=self.__balance: self.__balance-
=amt
        else: print('Auth failed or invalid')
    def bal(self,pin):
        if pin==self.__pin: return self.__balance
        else: return 'Auth failed'

atm=ATM('BankX','Main','1234',500); atm.deposit('1234',200);
atm.withdraw('0000',100); print('Balance:', atm.bal('1234'))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Auth failed or invalid
Balance: 700
```

7. ShoppingCart (items private)

Section: Encapsulation

Problem Statement:

Design a class ShoppingCart with customer_name as public, _cart_id as protected, and __items as private. Items should be added/removed only through methods.

Code:

```
class ShoppingCart:
    def __init__(self,name,cart_id):
        self.customer_name=name; self._cart_id=cart_id; self.__items=[]
    def add_item(self,item): self.__items.append(item)
    def remove_item(self,item):
        if item in self.__items: self.__items.remove(item)
    def items(self): return list(self.__items)

sc=ShoppingCart('Neha','C01'); sc.add_item('apple');
sc.add_item('bread'); sc.remove_item('apple')
print('Items:', sc.items())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Items: ['bread']
```

8. Patient (disease private)

Section: Encapsulation

Problem Statement:

Write a class Patient with patient_name as public, _age as protected, and __disease as private. Only allow doctors (via method) to update disease.

Code:

```
class Patient:
    def __init__(self,name,age): self.patient_name=name; self._age=age;
self.__disease=''
    def update_disease(self,role,d):
        if role=='doctor': self.__disease=d
    def disease(self): return self.__disease

p=Patient('Sam',30); p.update_disease('nurse','Flu');
p.update_disease('doctor','Cold'); print('Disease:', p.disease())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Disease: Cold
```

9. Course (students private)

Section: Encapsulation

Problem Statement:

Create a class Course where course_name is public, _course_code is protected, and __students is private. Students should be added/removed through methods only.

Code:

```
class Course:
    def __init__(self,name,code): self.course_name=name;
self._course_code=code; self.__students=[]
    def add_student(self,s): self.__students.append(s)
    def remove_student(self,s):
        if s in self.__students: self.__students.remove(s)
    def students(self): return list(self.__students)

c=Course('Math','M01'); c.add_student('A'); c.add_student('B');
c.remove_student('A'); print('Students:', c.students())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Students: ['B']
```

10. Loan (limit private)

Section: Encapsulation

Problem Statement:

Make a class Loan where borrower_name is public, _loan_type is protected, and __loan_amount is private. Loan amount must not exceed 1000000.

Code:

```
class Loan:
    def __init__(self,name,lt,amt=0): self.borrower_name=name;
self._loan_type=lt; self.__loan_amount=0; self.set_amount(amt)
    def set_amount(self,amt):
        if 0<=amt<=1000000: self.__loan_amount=amt
        else: print('Exceeds limit')
    def amount(self): return self.__loan_amount

ln=Loan('Raju','Home',500000); ln.set_amount(2000000); print('Loan:',
ln.amount())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Exceeds limit
Loan: 500000
```

11. Mobile (price private)

Section: Encapsulation

Problem Statement:

Write a class Mobile where brand is public, _model is protected, and __price is private. Allow price access only with getter and setter methods.

Code:

```
class Mobile:
    def __init__(self,brand,model,price): self.brand=brand;
self._model=model; self.__price=0; self.set_price(price)
    def set_price(self,p):
        if p>=0: self.__price=p
    def get_price(self): return self.__price

m=Mobile('Samsung','M12',15000); print('Price:', m.get_price())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Price: 15000
```

12. Wallet (balance private)

Section: Encapsulation

Problem Statement:

Create a class Wallet where owner is public, _wallet_id is protected, and __balance is private. Balance can only be modified using add_funds and spend_funds methods.

Code:

```
class Wallet:
    def __init__(self,owner,wid): self.owner=owner;
self._wallet_id=wid; self.__balance=0
    def add_funds(self,amt):
        if amt>0: self.__balance+=amt
    def spend_funds(self,amt):
        if 0<amt<=self.__balance: self.__balance-=amt
    def bal(self): return self.__balance

w=Wallet('Priya','W01'); w.add_funds(500); w.spend_funds(200);
print('Balance:', w.bal())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Balance: 300
```

13. Ticket (seat private)

Section: Encapsulation

Problem Statement:

Design a class Ticket with passenger_name as public, _flight_number as protected, and __seat_number as private. Seat number should be updated only by staff method.

Code:

```
class Ticket:
    def __init__(self,name,flight): self.passenger_name=name;
self._flight_number=flight; self.__seat_number=''
    def assign_seat(self,role,seat):
        if role=='staff': self.__seat_number=seat
    def seat(self): return self.__seat_number

t=Ticket('Gita','IN123'); t.assign_seat('user','12A');
t.assign_seat('staff','12A'); print('Seat:', t.seat())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Seat: 12A
```

14. Customer (credit_score private)

Section: Encapsulation

Problem Statement:

Make a class Customer with name as public, _customer_id as protected, and __credit_score as private. Credit score should be used internally for loan approval.

Code:

```
class Customer:
    def __init__(self,name,cid): self.name=name; self._customer_id=cid;
self.__credit_score=700
    def eligible(self): return self.__credit_score>650

cust=Customer('Maya','C100'); print('Eligible:', cust.eligible())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Eligible: True
```

15. Exam (result private)

Section: Encapsulation

Problem Statement:

Write a class Exam with subject as public, _exam_code as protected, and __result as private. Only a teacher method should update the result.

Code:

```
class Exam:
    def __init__(self,sub,code): self.subject=sub;
self._exam_code=code; self.__result=None
    def set_result(self,role,res):
        if role=='teacher': self.__result=res
    def result(self): return self.__result

ex=Exam('Physics','E01'); ex.set_result('student','Pass');
ex.set_result('teacher','Pass'); print('Result:', ex.result())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Result: Pass
```

16. SavingsAccount inherits BankAccount

Section: Inheritance

Problem Statement:

Write a program to create a class BankAccount with methods to deposit and withdraw money. Create a subclass SavingsAccount that adds a method to calculate interest.

Code:

```
class BankAccount:
    def __init__(self,balance=0): self.balance=balance
    def deposit(self,amt): self.balance+=amt
    def withdraw(self,amt):
        if amt<=self.balance: self.balance-=amt
```

```
    def show(self): print('Balance:', self.balance)

class SavingsAccount(BankAccount):
    def interest(self,rate): print(self.balance*rate/100)

s=SavingsAccount(1000); s.deposit(500); s.interest(5); s.show()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
75.0
Balance: 1500
```

17. Person -> Student

Section: Inheritance

Problem Statement:

Define a class Person with attributes name and age. Create a subclass Student that adds roll number and marks. Write a program to create an object of Student and display all details.

Code:

```
class Person:
    def __init__(self,name,age): self.name=name; self.age=age
class Student(Person):
    def __init__(self,name,age,roll,marks):
        super().__init__(name,age); self.roll=roll; self.marks=marks
    def show(self): print(self.name,self.age,self.roll,self.marks)

st=Student('Ishita',20,'R02',[80,85]); st.show()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Ishita 20 R02 [80, 85]
```

18. Book -> Magazine

Section: Inheritance

Problem Statement:

Create a class Book with attributes title and author. Derive a class Magazine that adds attributes issue number and month. Demonstrate inheritance by creating objects of both.

Code:

```
class Book:
    def __init__(self,title,author): self.title=title;
self.author=author
class Magazine(Book):
    def __init__(self,title,author,issue,month):
        super().__init__(title,author); self.issue=issue;
self.month=month
b=Book('B','A'); m=Magazine('M','Auth',10,'Nov');
print(b.title,m.title,m.issue,m.month)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
B M 10 Nov
```

19. Grandparent-Parent-Child

Section: Inheritance

Problem Statement:

Write a program with three classes: Grandparent, Parent, and Child. Each should have a method to introduce itself. Show how the child can access all methods.

Code:

```
class Grandparent:
    def intro(self): print('I am grandparent')
class Parent(Grandparent):
    def intro_p(self): print('I am parent')
class Child(Parent):
    def intro_c(self): print('I am child')

ch=Child(); ch.intro(); ch.intro_p(); ch.intro_c()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
I am grandparent
I am parent
I am child
```

20. Vehicle -> Car -> ElectricCar

Section: Inheritance

Problem Statement:

Define a class Vehicle with attributes brand and speed. Derive a class Car that adds fuel type, and then derive ElectricCar from Car that adds battery capacity. Demonstrate multilevel inheritance.

Code:

```
class Vehicle:
    def __init__(self,brand,speed): self.brand=brand; self.speed=speed
class Car(Vehicle):
    def __init__(self,brand,speed,fuel): super().__init__(brand,speed);
self.fuel=fuel
class ElectricCar(Car):
    def __init__(self,brand,speed,fuel,batt):
super().__init__(brand,speed,fuel); self.batt=batt

ec=ElectricCar('Tesla',120,'Electric',100);
print(ec.brand,ec.speed,ec.fuel,ec.batt)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Tesla 120 Electric 100
```

21. Employee->Manager->Director

Section: Inheritance

Problem Statement:

Create a class Employee with a salary attribute. Derive a class Manager that adds department, and then a class Director that adds decision-making methods. Show how inheritance works in the chain.

Code:

```
class Employee:
    def __init__(self,name,sal): self.name=name; self.sal=sal
class Manager(Employee):
    def __init__(self,name,sal,dept): super().__init__(name,sal);
self.dept=dept
class Director(Manager):
    def decide(self): print('Decision made by', self.name)

d=Director('R',100000,'Sales'); d.decide(); print(d.dept)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Decision made by R
Sales
```

22. Multiple inheritance - Sports & Music

Section: Inheritance

Problem Statement:

Write a program with a class Sports that has a method play(), and a class Music that has a method sing(). Create a class Student that inherits from both and demonstrate that a student can do both.

Code:

```
class Sports:
    def play(self): print('Playing sports')
class Music:
    def sing(self): print('Singing')
class Student(Sports,Music): pass
s=Student(); s.play(); s.sing()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Playing sports
Singing
```

23. Teacher multiple inheritance example

Section: Inheritance

Problem Statement:

Define two classes MathTeacher and ScienceTeacher, each with a teach() method. Create a Teacher class that inherits from both and demonstrate multiple inheritance.

Code:

```
class MathTeacher:
    def teach(self): print('Teaching Math')
class ScienceTeacher:
    def teach_sci(self): print('Teaching Science')
class Teacher(MathTeacher,ScienceTeacher): pass
t=Teacher(); t.teach(); t.teach_sci()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Teaching Math
Teaching Science
```

24. Phone & Camera -> SmartPhone

Section: Inheritance

Problem Statement:

Write a program with a class Phone (with a call method) and a class Camera (with a take_photo method). Create a SmartPhone class that inherits from both and can make calls and take photos.

Code:

```
class Phone:
    def call(self): print('Calling...')
class Camera:
    def take_photo(self): print('Photo taken')
class SmartPhone(Phone,Camera): pass
sp=SmartPhone(); sp.call(); sp.take_photo()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Calling...
Photo taken
```

25. Animal sound override

Section: Inheritance

Problem Statement:

Create a base class Animal with a method sound(). Derive classes Dog and Cat that override sound(). Demonstrate hierarchical inheritance by creating objects of both.

Code:

```
class Animal:
    def sound(self): print('Some sound')
class Dog(Animal):
    def sound(self): print('Bark')
class Cat(Animal):
    def sound(self): print('Meow')
Dog().sound(); Cat().sound()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Bark
Meow
```

26. Shape area override

Section: Inheritance

Problem Statement:

Define a base class Shape with a method area(). Derive classes Circle and Rectangle that implement their own area() methods. Demonstrate inheritance by calculating areas.

Code:

```
class Shape:
    def area(self): pass
class Circle(Shape):
    def __init__(self,r): self.r=r
    def area(self): print(3.14*self.r*self.r)
class Rectangle(Shape):
    def __init__(self,l,w): self.l=l; self.w=w
    def area(self): print(self.l*self.w)
Circle(3).area(); Rectangle(4,5).area()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
28.259999999999998
20
```

27. Calculator add (2 or 3 numbers)

Section: Polymorphism

Problem Statement:

Create a class Calculator with a method add() that can add two or three numbers.

Code:

```
class Calculator:
    def add(self,*args):
        print(sum(args))
c=Calculator(); c.add(2,3); c.add(1,2,3)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
5
6
```

28. Greet with optional message

Section: Polymorphism

Problem Statement:

Write a function greet() that prints 'Hello, <name>'. If an optional message is provided, it should print 'Hello, <name>, <message>'.

Code:

```
def greet(name,msg=None):
    if msg: print(f'Hello, {name}, {msg}')
    else: print(f'Hello, {name}')

greet('Sam'); greet('Sam','Good luck')
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Hello, Sam
Hello, Sam, Good luck
```

29. calculate_interest default rate

Section: Polymorphism

Problem Statement:

Write a method calculate_interest(amount, rate=5) that calculates simple interest. If no rate is given, use the default rate of 5%.

Code:

```
def calculate_interest(amount, rate=5):
    print((amount*rate)/100)

calculate_interest(1000); calculate_interest(1000,7)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
50.0
70.0
```

30. Student show_info optional age

Section: Polymorphism

Problem Statement:

Create a class Student with a method show_info() that prints the student's name and roll number. If age is provided, print that too.

Code:

```
class StudentPoly:
    def __init__(self,name,roll,age=None): self.name=name;
self.roll=roll; self.age=age
    def show_info(self):
        if self.age: print(self.name,self.roll,self.age)
        else: print(self.name,self.roll)
s1=StudentPoly('Nia','R10'); s2=StudentPoly('Oli','R11',20);
s1.show_info(); s2.show_info()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Nia R10
Oli R11 20
```

31. area square or rectangle

Section: Polymorphism

Problem Statement:

Write a method area() that calculates the area of a square when only one parameter is given, and the area of a rectangle when two parameters are given.

Code:

```
def area(*args):
    if len(args)==1: print(args[0]*args[0])
    elif len(args)==2: print(args[0]*args[1])

area(4); area(4,5)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
16
20
```

32. sum_numbers *args

Section: Polymorphism

Problem Statement:

Write a method sum_numbers(*args) that returns the sum of any number of integers passed to it.

Code:

```
def sum_numbers(*args): print(sum(args))
sum_numbers(1,2,3,4)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
10
```

33. multiply *args

Section: Polymorphism

Problem Statement:

Write a function multiply(*args) that calculates and returns the product of all given numbers.

Code:

```
def multiply(*args):
    p=1
    for x in args: p*=x
    print(p)
multiply(2,3,4)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
24
```

34. concat_strings *args

Section: Polymorphism

Problem Statement:

Write a function concat_strings(*args) that takes multiple strings and returns them joined together as one string.

Code:

```
def concat_strings(*args): print(''.join(args))
concat_strings('Hello',' ','World')
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Hello World
```

35. find_max *args

Section: Polymorphism

Problem Statement:

Write a function find_max(*args) that returns the largest number among the given inputs.

Code:

```
def find_max(*args): print(max(args))
find_max(3,9,1,6)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
9
```

36. calculate_bill *items

Section: Polymorphism

Problem Statement:

Write a function calculate_bill(*items) where each item represents a price. The function should return the total bill amount.

Code:

```
def calculate_bill(*items): print(sum(items))
calculate_bill(100,200,50)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
350
```

37. Shape draw override

Section: Polymorphism

Problem Statement:

Create a Shape base class with a draw() method. Override it in Circle and Square.

Code:

```
class Shape:
    def draw(self): print('Drawing shape')
class Circle(Shape):
    def draw(self): print('Drawing Circle')
class Square(Shape):
    def draw(self): print('Drawing Square')
Circle().draw(); Square().draw()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Drawing Circle
Drawing Square
```

38. Vehicle speed override

Section: Polymorphism

Problem Statement:

Create a Vehicle class with speed(). Override it in Car and Bike.

Code:

```
class Vehicle:
    def speed(self): print('Generic speed')
class Car(Vehicle):
    def speed(self): print('Car speed 120')
class Bike(Vehicle):
    def speed(self): print('Bike speed 80')
Car().speed(); Bike().speed()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Car speed 120
Bike speed 80
```

39. Employee salary override

Section: Polymorphism

Problem Statement:

Create an Employee class with salary(). Override it in Manager and Developer.

Code:

```
class EmployeeR:
    def salary(self): print('Base salary')
class Manager(EmployeeR):
    def salary(self): print('Manager salary 90000')
class Developer(EmployeeR):
    def salary(self): print('Developer salary 70000')
Manager().salary(); Developer().salary()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Manager salary 90000
Developer salary 70000
```

40. BankAccount interest override

Section: Polymorphism

Problem Statement:

Create a BankAccount class with interest_rate(). Override in SavingsAccount and CurrentAccount.

Code:

```
class BankAccR:
    def interest_rate(self): print('Base rate')
class SavingsAcc(BankAccR):
    def interest_rate(self): print('Savings 5%')
class CurrentAcc(BankAccR):
    def interest_rate(self): print('Current 3%')
SavingsAcc().interest_rate(); CurrentAcc().interest_rate()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Savings 5%
Current 3%
```

41. Bird fly override

Section: Polymorphism

Problem Statement:

Create a Bird class with fly(). Override it in Eagle and Penguin.

Code:

```
class Bird:
    def fly(self): print('Some fly')
class Eagle(Bird):
    def fly(self): print('Eagle soars')
class Penguin(Bird):
    def fly(self): print('Penguins cannot fly')
Eagle().fly(); Penguin().fly()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Eagle soars
Penguins cannot fly
```

42. Payment pay override

Section: Polymorphism

Problem Statement:

Create a Payment class with pay(). Override it in CreditCard and UPI.

Code:

```
class Payment:
    def pay(self): print('Paying')
class CreditCard(Payment):
    def pay(self): print('Paying by Card')
class UPI(Payment):
    def pay(self): print('Paying by UPI')
CreditCard().pay(); UPI().pay()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Paying by Card
Paying by UPI
```

43. Device start override

Section: Polymorphism

Problem Statement:

Create a Device class with start(). Override it in Laptop and Mobile.

Code:

```
class Device:
    def start(self): print('Device start')
class Laptop(Device):
    def start(self): print('Laptop starting')
class Mobile(Device):
    def start(self): print('Mobile starting')
Laptop().start(); Mobile().start()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Laptop starting
Mobile starting
```

44. Teacher teach override

Section: Polymorphism

Problem Statement:

Create a Teacher class with teach(). Override it in MathTeacher and ScienceTeacher.

Code:

```
class Teacher:
    def teach(self): print('Teaching')
class MathTeacher(Teacher):
    def teach(self): print('Teaching Math')
class ScienceTeacher(Teacher):
    def teach(self): print('Teaching Science')
MathTeacher().teach(); ScienceTeacher().teach()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Teaching Math
Teaching Science
```

45. Sports play override

Section: Polymorphism

Problem Statement:

Create a Sports base class with play(). Override it in Football and Cricket.

Code:

```
class SportsB:
    def play(self): print('Play sport')
class Football(SportsB):
    def play(self): print('Playing Football')
class Cricket(SportsB):
    def play(self): print('Playing Cricket')
Football().play(); Cricket().play()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Playing Football
Playing Cricket
```

46. Notification send override

Section: Polymorphism

Problem Statement:

Create a Notification class with send(). Override it in EmailNotification and SMSNotification.

Code:

```
class Notification:
    def send(self): print('Send generic')
class EmailNotification(Notification):
    def send(self): print('Send Email')
class SMSNotification(Notification):
    def send(self): print('Send SMS')
EmailNotification().send(); SMSNotification().send()
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Send Email
Send SMS
```

47. make_sound duck typing

Section: Polymorphism

Problem Statement:

Write a function make_sound(animal) that works for Dog and Duck classes (both have sound()).

Code:

```
def make_sound(animal): animal.sound()
class Dog:
    def sound(self): print('Woof')
class Duck:
    def sound(self): print('Quack')
make_sound(Dog()); make_sound(Duck())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Woof
Quack
```

48. start_engine duck typing

Section: Polymorphism

Problem Statement:

Create a function start_engine(vehicle) that works for Car and Bike classes (both have start() method).

Code:

```
def start_engine(v): v.start()
class CarD:
    def start(self): print('Car engine on')
class BikeD:
    def start(self): print('Bike engine on')
start_engine(CarD()); start_engine(BikeD())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Car engine on
Bike engine on
```

49. operate machine duck typing

Section: Polymorphism

Problem Statement:

Write a function operate(machine) that works for Printer and Scanner classes (both have run() method).

Code:

```
def operate(m): m.run()
class Printer:
    def run(self): print('Printing')
class Scanner:
    def run(self): print('Scanning')
operate(Printer()); operate(Scanner())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Printing
Scanning
```

50. play_game duck typing

Section: Polymorphism

Problem Statement:

Write a function play_game(player) that works for Footballer and Cricketer (both have play() method).

Code:

```
def play_game(p): p.play()
class Footballer:
    def play(self): print('Playing Football')
class Cricketer:
    def play(self): print('Playing Cricket')
play_game(Footballer()); play_game(Cricketer())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Playing Football
Playing Cricket
```

51. book_ticket duck typing

Section: Polymorphism

Problem Statement:

Write a function book_ticket(transport) that works for Bus and Train (both have book() method).

Code:

```
def book_ticket(t): t.book()
class Bus:
    def book(self): print('Booked Bus')
class Train:
    def book(self): print('Booked Train')
book_ticket(Bus()); book_ticket(Train())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Booked Bus
Booked Train
```

52. make_payment duck typing

Section: Polymorphism

Problem Statement:

Write a function make_payment(method) that works for CreditCard and PayPal (both have pay() method).

Code:

```
def make_payment(m): m.pay()
class CreditCard:
    def pay(self): print('Card paid')
class PayPal:
    def pay(self): print('PayPal paid')
make_payment(CreditCard()); make_payment(PayPal())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Card paid
PayPal paid
```

53. charge device duck typing

Section: Polymorphism

Problem Statement:

Write a function charge(device) that works for Phone and Laptop (both have charge() method).

Code:

```
def charge(d): d.charge()
class PhoneD:
    def charge(self): print('Phone charging')
class LaptopD:
    def charge(self): print('Laptop charging')
charge(PhoneD()); charge(LaptopD())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Phone charging
Laptop charging
```

54. study duck typing

Section: Polymorphism

Problem Statement:

Write a function study(student) that works for SchoolStudent and CollegeStudent (both have study() method).

Code:

```
def study(s): s.study()
class SchoolStudent:
    def study(self): print('Studying in school')
class CollegeStudent:
    def study(self): print('Studying in college')
study(SchoolStudent()); study(CollegeStudent())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Studying in school
Studying in college
```

55. draw duck typing

Section: Polymorphism

Problem Statement:

Write a function draw(shape) that works for Circle and Rectangle (both have draw() method).

Code:

```
def draw(s): s.draw()
class CircleD:
    def draw(self): print('Draw circle')
class RectangleD:
    def draw(self): print('Draw rectangle')
draw(CircleD()); draw(RectangleD())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Draw circle
Draw rectangle
```

56. open_file duck typing

Section: Polymorphism

Problem Statement:

Write a function open_file(filetype) that works for PDF and WordDoc (both have open() method).

Code:

```
def open_file(f): f.open()
class PDF:
    def open(self): print('Opening PDF')
class WordDoc:
    def open(self): print('Opening WordDoc')
open_file(PDF()); open_file(WordDoc())
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Opening PDF
Opening WordDoc
```

57. Point + operator

Section: Polymorphism

Problem Statement:

Create a Point class and overload + to add two points.

Code:

```
class Point:
    def __init__(self,x,y): self.x=x; self.y=y
    def __add__(self,other): return
Point(self.x+other.x,self.y+other.y)
    def __repr__(self): return f'Point({self.x},{self.y})'
p1=Point(1,2); p2=Point(3,4); print(p1+p2)
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Point(4,6)
```

58. Vector - operator

Section: Polymorphism

Problem Statement:

Create a Vector class and overload - to subtract two vectors.

Code:

```
class Vector:
    def __init__(self,x,y): self.x=x; self.y=y
    def __sub__(self,other): return Vector(self.x-other.x,self.y-
other.y)
    def __repr__(self): return f'Vector({self.x},{self.y})'
print(Vector(5,6)-Vector(2,3))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Vector(3,3)
```

59. Distance > operator

Section: Polymorphism

Problem Statement:

Create a Distance class and overload > to compare two distances.

Code:

```
class Distance:
    def __init__(self,m): self.m=m
    def __gt__(self,other): return self.m>other.m
print(Distance(10)>Distance(5))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
True
```

60. Student == operator

Section: Polymorphism

Problem Statement:

Create a Student class and overload == to compare marks of two students.

Code:

```
class Stud:
    def __init__(self,marks): self.marks=marks
    def __eq__(self,other): return self.marks==other.marks
print(Stud(80)==Stud(80)); print(Stud(80)==Stud(75))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
True
False
```

61. Time < operator

Section: Polymorphism

Problem Statement:

Create a Time class and overload < to compare two times.

Code:

```
class Time:
    def __init__(self,h,m): self.h=h; self.m=m
    def __lt__(self,other):
```

```
        return (self.h,self.m)<(other.h,other.m)
print(Time(10,30)<Time(11,0))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
True
```

62. ComplexNumber * operator

Section: Polymorphism

Problem Statement:

Create a ComplexNumber class and overload * to multiply two complex numbers.

Code:

```
class ComplexNumber:
    def __init__(self,a,b): self.a=a; self.b=b
    def __mul__(self,other): return ComplexNumber(self.a*other.a -
self.b*other.b, self.a*other.b + self.b*other.a)
    def __repr__(self): return f'({self.a}+{self.b}i)'
print(ComplexNumber(1,2)*ComplexNumber(3,4))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
(-5+10i)
```

63. Book + operator pages

Section: Polymorphism

Problem Statement:

Create a Book class and overload + to add pages of two books.

Code:

```
class BookOp:
    def __init__(self,pages): self.pages=pages
    def __add__(self,other): return BookOp(self.pages+other.pages)
    def __repr__(self): return f'Book({self.pages}p)'
print(BookOp(100)+BookOp(150))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
Book(250p)
```

64. BankAccount + operator merge balances

Section: Polymorphism

Problem Statement:

Create a BankAccount class and overload + to merge balances of two accounts.

Code:

```
class BankAcc:
    def __init__(self,balance): self.balance=balance
    def __add__(self,other): return BankAcc(self.balance+other.balance)
    def __repr__(self): return f'BankAcc({self.balance})'
print(BankAcc(500)+BankAcc(700))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
BankAcc(1200)
```

65. Fraction / operator

Section: Polymorphism

Problem Statement:

Create a Fraction class and overload / to divide two fractions.

Code:

```
class Fraction:
    def __init__(self,n,d): self.n=n; self.d=d
    def __truediv__(self,other): return Fraction(self.n*other.d,
self.d*other.n)
    def __repr__(self): return f'{self.n}/{self.d}'
print(Fraction(1,2)/Fraction(3,4))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
4/6
```

66. Rectangle == area compare

Section: Polymorphism

Problem Statement:

Create a Rectangle class and overload == to check if two rectangles have the same area.

Code:

```
class RectOp:
    def __init__(self,l,w): self.l=l; self.w=w
    def __eq__(self,other): return self.l*self.w==other.l*other.w
print(RectOp(2,6)==RectOp(3,4))
```

Output:

```
PS C:\Users\Mohammad Sayeed> python file.py
True
```