Task:

Duck Typing Tasks

1. Walk Like a Duck

Create two classes Duck and Person.

Both should have a method walk().

Write a function make_it_walk(obj) that accepts any object and calls walk().

Pass objects of both classes and observe.

```
class Duck:
    def walk(self):
        print('walks slowly')
class Person:
    def walk(self):
        print('Better walking style')
d1=Duck()
p1=Person()
def make_it_walk(obj):
    obj.walk()
    print('Their walking style')
make_it_walk(d1)
make_it_walk(p1)
```

2. Media Player Example

Create two classes:

MP3 → with method play()

Video → with method play()

Write a function start_media(obj) to call play() no matter the type.

```
lass MP3:
    def play(self):
        print('we can only listen the words')
class Video:
    def play(self):
        print('we can see the action')
m1=MP3()
v1=Video()
def start_media(obj):
    obj.play()
start_media(m1)
start_media(v1)
```

3. Payment System

Create two classes:

CreditCard → with method pay(amount)

UPI → with method pay(amount)

Write a function process_payment(obj, amount) to call pay().

```
class CreditCard:
    def pay(self,amount):
        print(f'paid Rs.{amount} using credit card')
class UPI:
    def pay(self,amount):
        print(f'paid Rs.{amount} using UPI')
def process_payment(obj,amount):
    obj.pay(amount)
c1=CreditCard()
u1=UPI()
process_payment(c1,1500)
process_payment(u1,2000)
```

Abstraction Tasks

4. Shape Area (Abstract)

Create an abstract class Shape with an abstract method area().

Subclasses:

Square → calculates side^2

Circle → calculates $\pi \times r^2$

```
class Shape:
    def area(self):
        pass
class Square(Shape):
    def area(self,side):
        print(f'Area of square for side {side} is {side*side}')
class Circle(Shape):
    def area(self,radius):
        pi=3.14
        print(f'Area of circle for radius {radius} is {pi*radius*radius} ')
shape=[Square(),Circle()]
for i in shape:
    i.area(10)
```

5. Vehicle Start (Abstract)

Create an abstract class Vehicle with an abstract method start().

Subclasses:

Car → prints "Car started"

Bike → prints "Bike started"

```
from abc import ABC,abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass
class Car(Vehicle):
    def start(self):
        print('Car Started')
class Bike(Vehicle):
    def start(self):
        print('Bike Started')
c1=Car()
c1.start()
b1=Bike()
b1.start()
```

6. Bank Account (Abstract)

Create an abstract class BankAccount with abstract method withdraw(amount).

Subclasses:

SavingsAccount → withdraw allowed if balance > 500

CurrentAccount → no minimum balance check

```
from abc import ABC,abstractmethod
class BankAccount(ABC):
    @abstractmethod
    def withdraw(self,amount):
        pass
class SavingsAccount(BankAccount):
    balance=15000
    def withdraw(self,amount):
        bal=self.balance-500
        if self.balance>500:
            print(f'withdraw allowed you can draw upto {bal}')
        else:
            print('withdraw not allowed')
class CurrentAccount(BankAccount):
    balance=2000
    def withdraw(self,amount):
        if self.balance==0:
            print('no minimum balance to check')
        else:
            print(f'your balance is {self.balance}')
s1=SavingsAccount()
s1.withdraw(2000)
c1=CurrentAccount()
c1.withdraw(15000)
```

7. Report Generation (Abstract)

Create an abstract class Report with abstract method generate().

Subclasses:

PDFReport → prints "PDF Report generated"

ExcelReport → prints "Excel Report generated"

```
from abc import ABC,abstractmethod
class Report(ABC):
    @abstractmethod
    def generate(self):
        pass
class PDFReport(Report):
    def generate(self):
        print('PDF Report generated')
class ExcelReport(Report):
    def generate(self):
        print('Excel Report generated')
```

```
reports=[PDFReport(),ExcelReport()]
for report in reports:
    report.generate()
```

8. Employee Work (Abstract)

Create an abstract class Employee with an abstract method work().

Subclasses:

Developer → prints "Writing code"

Tester → prints "Testing software"

```
class Employee:
    def work(self):
        pass
class Developer(Employee):
    def work(self):
        print('Writing code')
class Tester(Employee):
    def work(self):
        print('Testing Software')
employees=[Developer(),Tester()]
for employee in employees:
    employee.work()
```

9. Appliance Power (Abstract)

Create an abstract class Appliance with abstract method turn_on().

Subclasses:

Fan → prints "Fan is ON"

Light → prints "Light is ON"

```
from abc import ABC,abstractmethod
class Appliance(ABC):
    @abstractmethod
    def turn_on(self):
        pass
class Fan(Device):
    def turn_on(self):
        print('Fan is ON')
class Light(Device):
    def turn_on(self):
        print('Light is ON')
devices=[Fan(),Light()]
for device in devices:
    device.turn_on()
```