Oops

1.Identify the class names from the following:

joiningDate

Customer

salary

age

hungry

running

excellent

Book

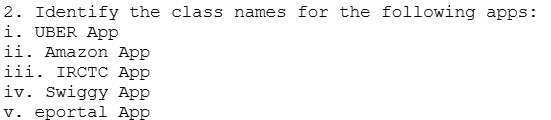
Car

transferMoney

MoneyTransfer

Sol : Customer Book Car MoneyTransfer

2.

. 

Sol: i. UBER App

Class Names:

* Driver
* Passenger
* Payment
* Vehicle

ii. Amazon App

Class Names:

* Customer
* Product
* Order
* Cart

iii. IRCTC App

Possible Class Names:

* User
* Train
* Ticket
* Reservation
* Station

iv. Swiggy App

Class Names:

* Restaurant
* MenuItem
* Order
* DeliveryAgent
* Customer
* Address
* Payment

v. ePortal App (e.g., college or corporate portal)

Possible Class Names:

* Student / Employee
* Course / Project
* Assignment / Task
* Attendance
* Login
* LeaveRequest

**3.Create a class Car with attributes brand, model, and year. Instantiate an object of this class and print its details.**

class Car:

def \_\_init\_\_(self, brand, model, year):

self.brand = brand

self.model = model

self.year = year

def display\_details(self):

print(f"Car Details:\nBrand: {self.brand}\nModel: {self.model}\nYear: {self.year}")

# Instantiate a Car object and print details

if \_\_name\_\_ == "\_\_main\_\_":

my\_car = Car("Toyota", "Corolla", 2022)

my\_car.display\_details()

**4.Write a class Rectangle that takes length and breadth as parameters. Create a method to calculate the area of the rectangle. Instantiate two rectangles and compare their areas.**

class Rectangle:

def \_\_init\_\_(self, length, breadth):

self.length = length

self.breadth = breadth

def area(self):

return self.length \* self.breadth

def \_\_str\_\_(self):

return f"Rectangle [Length: {self.length}, Breadth: {self.breadth}, Area: {self.area()}]"

from basic\_concepts.rectangle import Rectangle

if \_\_name\_\_ == "\_\_main\_\_":

rect1 = Rectangle(10, 5)

rect2 = Rectangle(7, 8)

print(rect1)

print(rect2)

if rect1.area() > rect2.area():

print("Rectangle 1 larger area.")

elif rect2.area() > rect1.area():

print("Rectangle 2 larger area.")

else:

print("Both rectangles have equal area.")

**5.Create a class Student with attributes name, roll\_number, and marks. Define a method to display the student's details. Instantiate the class and print the details of a student.**

class Student:

def \_\_init\_\_(self, name, roll\_number, marks):

self.name = name

self.roll\_number = roll\_number

self.marks = marks

def \_\_str\_\_(self):

return f"Student [Name: {self.name}, Roll Number: {self.roll\_number}, Marks: {self.marks}]"

from basic\_concepts.student import Student

if \_\_name\_\_ == "\_\_main\_\_":

student1 = Student("Priyanka", "21BCE8358", 21)

print(student1)

**6.Create a class BankAccount with a private attribute \_balance. Implement methods to deposit and withdraw money while ensuring the balance never goes negative.**

**7.Modify the BankAccount class to include a getter method for checking the balance and a setter method to update it (ensuring balance cannot be set to a negative value).**

class BankAccount:  
 def \_\_init\_\_(self, initial\_balance=0):  
 self.\_balance = initial\_balance  
  
 def deposit(self, amount):  
 if amount > 0:  
 self.\_balance += amount  
 print(f"deposit {amount}")  
 else:  
 print("deposit must be positive")  
  
 def withdraw(self, amount):  
 if amount <= self.\_balance:  
 self.\_balance -= amount  
 else:  
 print("insufficient amount !")  
  
 def \_\_str\_\_(self):  
 return f"current balance = {self.\_balance}"  
  
 #getter method  
 def get\_balance(self):  
 return self.\_balance  
  
 #setter method  
 def set\_balance(self , new\_balance):  
 if new\_balance >= 0:  
 self.\_balance = new\_balance  
 print(f"the balance updated to new balance {new\_balance}")  
 else:  
 print("balance cannot set to negative !")

from entity.bankAccount import BankAccount  
if \_\_name\_\_ == "\_\_main\_\_":  
 Account = BankAccount(500)  
 print(Account)  
 Account.deposit(200)  
 print(f" Total balance after deposit : {Account}")  
 Account.withdraw(100)  
 print(f" Total balance after withdrawl : {Account}")  
 print(Account)  
 Account.withdraw(1500)  
 #check balance through get balance  
 print("Balance : " ,Account.get\_balance())  
 #set balance through set balance  
 Account.set\_balance(2000)  
 print(Account)

**8.Create a class Laptop with private attributes brand and price. Provide getter and setter methods to access and modify these attributes**.

class Laptop:

def \_\_init\_\_(self, brand, price):

self.\_\_brand = brand

self.\_\_price = price

def get\_brand(self):

return self.\_\_brand

def set\_brand(self, brand):

self.\_\_brand = brand

def get\_price(self):

return self.\_\_price

def set\_price(self, price):

if price > 0:

self.\_\_price = price

else:

print("Invalid price. Must be greater than 0.")

def \_\_str\_\_(self):

return f"Laptop [Brand: {self.\_\_brand}, Price: ₹{self.\_\_price}]"

from encapsulation.laptop import Laptop

if \_\_name\_\_ == "\_\_main\_\_":

laptop1 = Laptop("HP", 55000)

# Access via getter

print("Brand:", laptop1.get\_brand())

print("Price:", laptop1.get\_price())

# Modify via setter

laptop1.set\_price(60000)

laptop1.set\_brand("Dell")

# Display full details

print(laptop1)

**9.Create a class Animal with a method make\_sound(). Derive subclasses Dog and Cat from it, and override make\_sound() to print respective sounds.**

class Animal:

def make\_sound(self):

print("Some generic animal sound")

class Dog(Animal):

def make\_sound(self):

print("Bark! Bark!")

class Cat(Animal):

def make\_sound(self):

print("Meow! Meow!")

from inheritance.animal import Dog, Cat

if \_\_name\_\_ == "\_\_main\_\_":

d = Dog()

c = Cat()

d.make\_sound()

c.make\_sound()

**10.Create a base class Employee with attributes name and salary. Derive two classes: Manager and Developer, where Manager has an additional attribute team\_size and Developer has programming\_language.**

class Employee:

def \_\_init\_\_(self, name, salary):

self.name = name

self.salary = salary

def \_\_str\_\_(self):

return f"Employee [Name: {self.name}, Salary: ₹{self.salary}]"

class Manager(Employee):

def \_\_init\_\_(self, name, salary, team\_size):

super().\_\_init\_\_(name, salary)

self.team\_size = team\_size

def \_\_str\_\_(self):

return f"Manager [Name: {self.name}, Salary: ₹{self.salary}, Team Size: {self.team\_size}]"

class Developer(Employee):

def \_\_init\_\_(self, name, salary, programming\_language):

super().\_\_init\_\_(name, salary)

self.programming\_language = programming\_language

def \_\_str\_\_(self):

return f"Developer [Name: {self.name}, Salary: ₹{self.salary}, Language: {self.programming\_language}]"

from inheritance.employee import Manager, Developer

def main():

mgr = Manager("Priya", 80000, 5)

dev = Developer("Rahul", 60000, "Python")

print(mgr)

print(dev)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**11.Create a class Shape with a method area(). Inherit Circle and Square classes from Shape and implement their respective area calculations.**

import math

class Shape:

def area(self):

raise NotImplementedError("Subclass must implement abstract method")

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def area(self):

return math.pi \* self.radius \* self.radius

def \_\_str\_\_(self):

return f"Circle [Radius: {self.radius}, Area: {self.area():.2f}]"

class Square(Shape):

def \_\_init\_\_(self, side):

self.side = side

def area(self):

return self.side \* self.side

def \_\_str\_\_(self):

return f"Square [Side: {self.side}, Area: {self.area()}]"

from inheritance.shape import Circle, Square

if \_\_name\_\_ == "\_\_main\_\_":

c = Circle(5)

s = Square(6)

print(c)

print(s)

**12.Create a class Vehicle with attributes brand and speed. Derive Car and Bike classes from it and add specific attributes and methods to each.**

class Vehicle:

def \_\_init\_\_(self, brand, speed):

self.brand = brand

self.speed = speed

def \_\_str\_\_(self):

return f"Vehicle [Brand: {self.brand}, Speed: {self.speed} km/h]"

class Car(Vehicle):

def \_\_init\_\_(self, brand, speed, fuel\_type):

super().\_\_init\_\_(brand, speed)

self.fuel\_type = fuel\_type

def \_\_str\_\_(self):

return f"Car [Brand: {self.brand}, Speed: {self.speed} km/h, Fuel: {self.fuel\_type}]"

class Bike(Vehicle):

def \_\_init\_\_(self, brand, speed, cc):

super().\_\_init\_\_(brand, speed)

self.cc = cc

def \_\_str\_\_(self):

return f"Bike [Brand: {self.brand}, Speed: {self.speed} km/h, Engine: {self.cc}cc]"

from inheritance.vehicle import Car, Bike

def main():

c1 = Car("Toyota", 180, "Petrol")

b1 = Bike("Yamaha", 140, 150)

print(c1)

print(b1)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**13.Modify the Shape class hierarchy to include a display() method that prints shape details. Override this method in Circle and Square**.

# Q13: Shape class with display() method overridden in Circle and Square

import math

class Shape:

def display(self):

print("This is a generic shape.")

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.radius = radius

def display(self):

print(f"Circle → Radius: {self.radius}, Area: {math.pi \* self.radius \*\* 2:.2f}")

class Square(Shape):

def \_\_init\_\_(self, side):

self.side = side

def display(self):

print(f"Square → Side: {self.side}, Area: {self.side \*\* 2}")

from polymorphism.shape\_display import Circle, Square

if \_\_name\_\_ == "\_\_main\_\_":

c = Circle(5)

s = Square(4)

c.display()

s.display()

**14.Create a function calculate\_total\_price() that takes different objects (Laptop, Mobile, Tablet) and calls their method get\_price() to calculate the total price dynamically.**

# Q14: Demonstrate polymorphism with get\_price() in multiple classes

class Laptop:

def \_\_init\_\_(self, price):

self.price = price

def get\_price(self):

return self.price

class Mobile:

def \_\_init\_\_(self, price):

self.price = price

def get\_price(self):

return self.price

class Tablet:

def \_\_init\_\_(self, price):

self.price = price

def get\_price(self):

return self.price

from polymorphism.electronics import Laptop, Mobile, Tablet

#this is main method

def calculate\_total\_price(items):

total = 0

for item in items:

total += item.get\_price() # Polymorphic method call

return total

if \_\_name\_\_ == "\_\_main\_\_":

items = [

Laptop(55000),

Mobile(22000),

Tablet(18000),

Laptop(45000),

Mobile(26000)

]

total = calculate\_total\_price(items)

print(f"Total Price of All Items: ₹{total}")

**15.Create a class Person with a method introduce(). Override it in subclasses Teacher and Student to print different introductions.**

class Person:

def introduce(self):

print("Hi, I'm a person.")

class Teacher(Person):

def introduce(self):

print("Hi, I'm a teacher. I love teaching.")

class Student(Person):

def introduce(self):

print("Hi, I'm a student. I love learning.")

from polymorphism.person import Teacher, Student

if \_\_name\_\_ == "\_\_main\_\_":

t = Teacher()

s = Student()

t.introduce()

s.introduce()

**16.Overload the \_\_str\_\_ method in a class Bank to return a formatted string when the object is printed.**

# Q16: Overload \_\_str\_\_ method in Bank class to print formatted output

class Bank:

def \_\_init\_\_(self, name, branch, ifsc\_code):

self.name = name

self.branch = branch

self.ifsc\_code = ifsc\_code

def \_\_str\_\_(self):

return (

f"🏦 Bank Details\n"

f"---------------------\n"

f"Name : {self.name}\n"

f"Branch : {self.branch}\n"

f"IFSC Code : {self.ifsc\_code}"

)

from polymorphism.bank\_str import Bank

if \_\_name\_\_ == "\_\_main\_\_":

bank1 = Bank("HDFC Bank", "Anna Nagar", "HDFC0001234")

bank2 = Bank("ICICI Bank", "T. Nagar", "ICIC0005678")

print(bank1)

print()

print(bank2)

**17.Create an abstract class Appliance with an abstract method turn\_on(). Implement subclasses WashingMachine and Refrigerator with specific functionality.**

# Q17: Abstract Appliance class with turn\_on()

from abc import ABC, abstractmethod

class Appliance(ABC):

@abstractmethod

def turn\_on(self):

pass

class WashingMachine(Appliance):

def turn\_on(self):

print("Washing Machine is now running.")

class Refrigerator(Appliance):

def turn\_on(self):

print("Refrigerator is now cooling.")

from abstraction.appliance import WashingMachine, Refrigerator

if \_\_name\_\_ == "\_\_main\_\_":

wm = WashingMachine()

fridge = Refrigerator()

wm.turn\_on()

fridge.turn\_on()

**18.Create an abstract class Animal with an abstract method move(). Implement Bird and Fish subclasses with their movement styles.**

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def move(self):

pass

class Bird(Animal):

def move(self):

print("Bird flies in the sky.")

class Fish(Animal):

def move(self):

print("Fish swims in the water.")

from abstraction.animal\_move import Bird, Fish

if \_\_name\_\_ == "\_\_main\_\_":

b = Bird()

f = Fish()

b.move()

f.move()

**19.Create an abstract class Payment with an abstract method process\_payment(). Implement CreditCardPayment and PayPalPayment subclasses.**

from abc import ABC, abstractmethod

class Payment(ABC):

@abstractmethod

def process\_payment(self, amount):

pass

class CreditCardPayment(Payment):

def process\_payment(self, amount):

print(f"Processing ₹{amount} via Credit Card.")

class PayPalPayment(Payment):

def process\_payment(self, amount):

print(f"Processing ₹{amount} via PayPal.")

from abstraction.payment import CreditCardPayment, PayPalPayment

if \_\_name\_\_ == "\_\_main\_\_":

cc = CreditCardPayment()

paypal = PayPalPayment()

cc.process\_payment(1000)

paypal.process\_payment(1500)

**20. Shop has products , implement the requirement using collections**

class Shop:

def \_\_init\_\_(self, name):

self.name = name

self.products = [] # list of dictionaries

def add\_product(self, product\_id, name, price):

product = {

"product\_id": product\_id,

"name": name,

"price": price

}

self.products.append(product)

def display\_products(self):

print(f"Products in {self.name}:")

for product in self.products:

print(f"ID: {product['product\_id']} | Name: {product['name']} | Price: ₹{product['price']}")

from collections\_examples.shop import Shop

if \_\_name\_\_ == "\_\_main\_\_":

my\_shop = Shop("Tech Store")

my\_shop.add\_product(101, "Mouse", 500)

my\_shop.add\_product(102, "Keyboard", 800)

my\_shop.add\_product(103, "Monitor", 7000)

my\_shop.display\_products()

**21. Company has employees , implement the requirement using collections**

class Employee:

def \_\_init\_\_(self, emp\_id, name, role):

self.emp\_id = emp\_id

self.name = name

self.role = role

def \_\_str\_\_(self):

return f"Employee [ID: {self.emp\_id}, Name: {self.name}, Role: {self.role}]"

class Company:

def \_\_init\_\_(self, name):

self.name = name

self.employees = {} # key = emp\_id, value = Employee object

def add\_employee(self, emp):

self.employees[emp.emp\_id] = emp

def display\_employees(self):

print(f"Employees of {self.name}:")

for emp in self.employees.values():

print(emp)

from collections\_examples.company import Company, Employee

if \_\_name\_\_ == "\_\_main\_\_":

my\_company = Company("CodeWave Pvt Ltd")

emp1 = Employee(1, "Amit", "Developer")

emp2 = Employee(2, "Sneha", "Manager")

emp3 = Employee(3, "Priya", "Analyst")

my\_company.add\_employee(emp1)

my\_company.add\_employee(emp2)

my\_company.add\_employee(emp3)

my\_company.display\_employees()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_python task \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Accept a date and the number of days to add it and print the resultant date.

from datetime import datetime, timedelta

# Import the necessary tools from datetime module

from datetime import datetime, timedelta

# Ask the user to enter a date

date\_str = input("Enter a date (in YYYY-MM-DD format): ")

# Ask the user to enter how many days to add

days = int(input("Enter number of days to add: "))

# Convert the date string to a datetime object

given\_date = datetime.strptime(date\_str, "%Y-%m-%d")

# Add the given number of days to the date

new\_date = given\_date + timedelta(days=days)

# Print the new date in the same format

print("New Date after adding days is:", new\_date.strftime("%Y-%m-%d"))

2. Accept a 5-digit decimal number as input and display the sum of the digits as output.

# Ask the user to enter a 5-digit decimal number

num = input("Enter a 5-digit decimal number: ")

# Remove the decimal point

num = num.replace(".", "")

# Initialize the sum

digit\_sum = 0

# Loop through each character and add digits

for digit in num:

digit\_sum += int(digit)

# Print the result

print("Sum of digits is:", digit\_sum)

3. An employee is considered for on-site depending on these conditions

(1) An employee Should have Passport

(ii) Communication should be good

(iii) His training feedback should be good

(iv) Should be at-least 2years experienced.

(v) Age should be greater than or equal to 23. Using above conditions, check if an employee is eligible to go to on-site or not.

def check\_onsite\_eligibility(passport, communication, training, experience, age):

if passport == "yes":

if communication == "yes":

if training == "yes":

if experience >= 2:

if age >= 23:

print("You are eligible for on-site.")

else:

print("You are NOT eligible for on-site (Age must be >= 23).")

else:

print("You are NOT eligible for on-site (Experience must be >= 2 years).")

else:

print("You are NOT eligible for on-site (Training feedback not good).")

else:

print("You are NOT eligible for on-site (Poor communication skills).")

else:

print("You are NOT eligible for on-site (No passport).")

# Collect inputs from user

passport = input("Do you have a passport? (yes/no): ")

communication = input("Is your communication good? (yes/no): ")

training = input("Is your training feedback good? (yes/no): ")

experience = int(input("Enter your years of experience: "))

age = int(input("Enter your age: "))

# Call the function with inputs

check\_onsite\_eligibility(passport, communication, training, experience, age)

4. Calculate electricity bill for following constraints.

(i) If units exceed 1000, then charge Rs. 10/- per unit.

(ii) If units exceed 500, then charge Rs.5/- per unit.

(iii) If units exceed 200, then charge Rs.2/- per unit.

(iv) in other cases, charge Rs. 1/- per unit.

def calculate\_electricity\_bill(units):

if units > 1000:

bill = units \* 10

elif units > 500:

bill = units \* 5

elif units > 200:

bill = units \* 2

else:

bill = units \* 1

print(f"Your electricity bill is: Rs.{bill}")

# Take input from the user

units\_consumed = int(input("Enter number of electricity units consumed: "))

# Call the function

calculate\_electricity\_bill(units\_consumed)

5. Implement the following logic to arrive at the loan eligibility for an employee:

(a) For an unmarried permanent job holder, if the service has been for 30 years or more, the loan amount is Rs.50,000; otherwise, the loan amount is Rs.25,000

(b) For a married permanent job holder, if the service has been for 30 years or more, the loan amount is Rs.60,000; otherwise, the loan amount is Rs.35,000

(c) For temporary job holder, the loan amount is Rs. 10,000 Test the application using the below given test cases

(a) a married temporary job holder

(b) an unmarried temporary job holder

(c) a temporary job holder with 30 years of service

(d) a married permanent job holder with 30 years of service

(e) an unmarried permanent job holder with 30 years' service

(f) a married permanent job holder with 25 years' service.

def check\_loan\_eligibility():

marital\_status = input("Enter marital status (married/unmarried): ").lower()

job\_type = input("Enter job type (permanent/temporary): ").lower()

service\_years = int(input("Enter years of service: "))

loan = 0 # initialize loan variable

if job\_type == "temporary":

loan = 10000

else:

if marital\_status == "unmarried":

if service\_years >= 30:

loan = 50000

else:

loan = 25000

elif marital\_status == "married":

if service\_years >= 30:

loan = 60000

else:

loan = 35000

print("Loan eligible: Rs.", loan)

# Call the function

check\_loan\_eligibility()

6. If in a year January 1 st is Sunday, then list all the dates of Sunday in that month.

def list\_sundays\_in\_january():

from datetime import date, timedelta

year = int(input("Enter the year: "))

# Jan 1st of the given year

start\_date = date(year, 1, 1)

# Check if Jan 1st is Sunday (weekday() == 6 means Sunday)

if start\_date.weekday() == 6:

current\_date = start\_date

print(f"Sundays in January {year}:")

while current\_date.month == 1:

print(current\_date.strftime("%Y-%m-%d"))

current\_date += timedelta(days=7)

else:

print("January 1st is not a Sunday in this year.")

# Call the function

list\_sundays\_in\_january()

7. Print Series

1/2, 2/5, 3/10, 4/17, 5/26, 6/37,.............

def print\_series(n\_terms):

numerator = 1

denominator = 2

print("Series:")

for i in range(n\_terms):

print(f"{numerator}/{denominator}", end=", " if i < n\_terms - 1 else "\n")

numerator += 1

denominator += (2 \* i + 3) # Increase with next odd number

# Example: Print first 10 terms

print\_series(10)

0,3,8,15,24, 35, 48, 63,….

def generate\_series\_1(n\_terms):

print("Pattern 1:")

for i in range(n\_terms):

value = i \* (i + 2)

print(value, end=", " if i < n\_terms - 1 else "\n")

generate\_series\_1(8)

8. Find the sum of the elements of the array.

def sum\_of\_array():

# Ask the user to enter numbers separated by spaces

user\_input = input("Enter numbers separated by space: ")

# Convert the input string to a list of integers using map()

numbers = list(map(int, user\_input.split()))

# Calculate the sum of elements

total = sum(numbers)

# Display the result

print(f"Sum of the array elements: {total}")

# Call the function

sum\_of\_array()

#another way

def sum\_of\_array():

user\_input = input("Enter numbers separated by space: ")

numbers = list(map(int, user\_input.split()))

total = 0 # Initialize sum

for num in numbers:

total += num # Add each element to total

print(f"Sum of the array elements: {total}")

# Call the function

sum\_of\_array()

#another way list comprehension

def sum\_of\_array():

total = sum([int(x) for x in input("Enter numbers separated by space: ").split()])

print(f"Sum of the array elements: {total}")

# Call the function

sum\_of\_array()

9. Read a sentence. Find a word and replace it with another word. (Note: Both words should be same in length)

def replace\_word\_same\_length():

sentence = input("Enter a sentence: ")

old\_word = input("Enter the word to replace: ")

new\_word = input("Enter the new word (same length): ")

if len(old\_word) != len(new\_word):

print("Both words must be of the same length.")

exit()

if old\_word not in sentence:

print("The word to be replaced was not found in the sentence.")

exit()

updated\_sentence = sentence.replace(old\_word, new\_word)

print("Updated Sentence:", updated\_sentence)

# Call the function

replace\_word\_same\_length()

10. Create a function that takes a sentence as input and does the following validation

(a) Length of the sentence must me minimum 10

(b) Sentence should have at-least one space.

(c) Sentence should not have dot in between.

(d) Should not start and end with spaces

(e) Must end with dot

def validate\_sentence():

sentence = input("Enter a sentence: ")

if len(sentence) >= 10:

if ' ' in sentence:

if not sentence.startswith(' ') and not sentence.endswith(' '):

if sentence.endswith('.'):

if '.' not in sentence[:-1]:

print("✅ Sentence is valid.")

else:

print("❌ Dot should not appear in the middle of the sentence.")

else:

print("❌ Sentence must end with a dot.")

else:

print("❌ Sentence should not start or end with a space.")

else:

print("❌ Sentence must contain at least one space.")

else:

print("❌ Sentence must be at least 10 characters long.")

# Call the function

validate\_sentence()

11. Write a function validatel Paddress that accepts an ipAddress (string) as an input and does the following validations

(a) Length should be minimum 7 and maximum 15

(b) Should not start and end with dot

(c) Should have exactly 3 dots

(d) Dots should not be in consecutive positions.

(e) Should not permit alphabets and special characters

def validate\_ip\_address():

ip = input("Enter the IP address: ")

if 7 <= len(ip) <= 15:

if not (ip.startswith('.') or ip.endswith('.')):

if ip.count('.') == 3:

if '..' not in ip:

parts = ip.split('.')

all\_numeric = True

for part in parts:

if not part.isdigit():

all\_numeric = False

break

if all\_numeric:

print("✅ Valid IP address.")

else:

print("❌ IP address should contain only numbers and dots.")

else:

print("❌ IP address should not have consecutive dots.")

else:

print("❌ IP address should contain exactly 3 dots.")

else:

print("❌ IP address should not start or end with a dot.")

else:

print("❌ Length of IP address must be between 7 and 15 characters.")

# Call the function

validate\_ip\_address()

12. Provide logic to validate a given a password whether the password is valid or not

(a) Length should be minimum 10 characters and maximum 50 characters

(b) Should not start and end with spaces

(c) Should not contain any special character other than underscore (\_)

import string

def validate\_password(password):

if len(password) < 10:

return "The password should be minimum 10 characters."

if len(password) > 50:

return "The password should be maximum 50 characters."

if password.startswith(" "):

return "The password should not start with space."

if password.endswith(" "):

return "The password should not end with space."

allowed\_chars = string.ascii\_letters + string.digits + '\_'

for i in password:

if i not in allowed\_chars:

return "Only letters, digits, and underscore (\_) are allowed."

return "The password is valid."

# Input from the user

password = input("Enter a password for validation: ")

print(validate\_password(password))

File Handling

File Handling Exercise:

----------------------------------

1.Filter Products by Category and Write to New File

Write a Python program to read product.xlsx and filter the products based on a specified category (e.g., "Electronics"). Write the filtered products into a new file, filtered\_products.xlsx. If the specified category doesn't exist in the dataset, raise a custom exception CategoryNotFoundException and log the error.

import openpyxl

import logging

# Set up logging configuration

# This will create a log file to store any error messages if the category is not found

logging.basicConfig(filename='category\_filter\_log.txt', level=logging.ERROR,

format='%(asctime)s - %(levelname)s - %(message)s')

# Define a custom exception to handle category not found situations

class CategoryNotFoundException(Exception):

pass

# Load the existing Excel workbook containing products

workbook = openpyxl.load\_workbook(r"D:\product (2).xlsx")

# Select the active sheet (or use workbook["SheetName"] if you know the exact name)

product\_sheet = workbook.active

# Create a new workbook to store the filtered data

filtered\_workbook = openpyxl.Workbook()

filtered\_sheet = filtered\_workbook.active

filtered\_sheet.title = "filtered\_products"

# Set the category you want to filter by

category\_to\_filter = "Electronics"

# Copy the header row from the source sheet to the new sheet

for col in range(1, product\_sheet.max\_column + 1):

filtered\_sheet.cell(row=1, column=col).value = product\_sheet.cell(row=1, column=col).value

# Boolean flag to check if any matching category was found

found = False

# Start writing filtered data from the second row

new\_row = 2

# Loop through each row of the product sheet, starting from row 2 to skip header

for row in range(2, product\_sheet.max\_row + 1):

# Read the category value (assuming it is in column 3)

category = product\_sheet.cell(row=row, column=3).value

# Check if the category matches the one we're filtering

if category == category\_to\_filter:

found = True # Mark that we found at least one match

# Copy all column values from this row to the new sheet

for col in range(1, product\_sheet.max\_column + 1):

filtered\_sheet.cell(row=new\_row, column=col).value = product\_sheet.cell(row=row, column=col).value

new\_row += 1 # Move to the next row in the filtered sheet

# If no matching category was found, raise and log an error

if not found:

error\_message = f"Category '{category\_to\_filter}' not found in the dataset."

logging.error(error\_message)

raise CategoryNotFoundException(error\_message)

# Save the new workbook containing filtered data

filtered\_workbook.save(r"D:\gender\_wise\_employee.xlsx")

print("Task completed.")

# Close both the original and new workbooks to release resources

workbook.close()

filtered\_workbook.close()

2. Append New Products to the File

Allow users to add a new product (Product ID, Name, Category, Price, Stock) to product.xlsx. If the product ID already exists, raise a DuplicateProductException.

from openpyxl import load\_workbook

import logging

# Setup logging to store error messages in a file

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Define custom exception for duplicate product

class DuplicateProductException(Exception):

pass

# Function to append new product

def append\_new\_product(file\_path, product\_id, name, category, price, stock):

try:

# Load the workbook and select active sheet

wb = load\_workbook("D:\product\_with\_stock.xlsx")

ws = wb.active

# Get index of Product\_ID column from headers

headers = [cell.value for cell in ws[1]]

id\_index = headers.index("Product\_ID") + 1 # 1-based index

# Check if Product\_ID already exists

for row in ws.iter\_rows(min\_row=2, values\_only=True):

if row[id\_index - 1] == product\_id:

raise DuplicateProductException(f"Product ID {product\_id} already exists.")

# Append the new product details

ws.append([product\_id, name, category, price, stock])

# Save workbook

wb.save(file\_path)

print(f"Product '{name}' added successfully.")

except DuplicateProductException as e:

logging.error(str(e))

print(str(e))

except Exception as e:

logging.error("Unexpected error: " + str(e))

print("Unexpected error:", str(e))

# Sample usage

append\_new\_product(

r"D:\appended\_new\_product.xlsx",

107, "Smartwatch", "Electronics", 3500, 20

)

3. Read Employee Details and Validate Salary

Read employee.xlsx and check if any employee has a salary below the minimum threshold (e.g., 10000). Raise a LowSalaryException for such cases and log the errors into error\_log.txt.

from openpyxl import load\_workbook

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Define custom exception

class LowSalaryException(Exception):

pass

# Function to validate employee salaries

def validate\_employee\_salary(file\_path, min\_salary=10000):

try:

# Load the workbook and get the sheet

wb = load\_workbook(file\_path)

ws = wb["Employees"] # assuming the sheet name is "Employees"

# Get the column index for Salary

headers = [cell.value for cell in ws[1]]

salary\_index = headers.index("Salary") + 1 # Convert to 1-based index

# Iterate through employee records and check salary

for row in ws.iter\_rows(min\_row=2, values\_only=True):

employee\_id, name, salary, email = row

if salary < min\_salary:

# Raise and log if salary is below threshold

raise LowSalaryException(f"Employee {name} (ID: {employee\_id}) has low salary: {salary}")

print("All employee salaries are valid.")

except LowSalaryException as e:

logging.error(str(e))

print("Low salary found:", e)

except Exception as e:

logging.error("Unexpected error: " + str(e))

print("Unexpected error:", str(e))

# Run the validation

validate\_employee\_salary("employee.xlsx")

4. Update Customer Email Address

Read customer.xlsx and update the email address for a given Customer ID. If the Customer ID does not exist, raise a CustomerNotFoundException.

from openpyxl import load\_workbook

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Custom exception for customer not found

class CustomerNotFoundException(Exception):

pass

# Function to update customer email

def update\_customer\_email(file\_path, customer\_id, new\_email):

try:

# Load the workbook and sheet

wb = load\_workbook(file\_path)

ws = wb["Customers"] # assuming the sheet is named "Customers"

# Get headers and Customer\_ID column index

headers = [cell.value for cell in ws[1]]

id\_index = headers.index("Customer\_ID") + 1

email\_index = headers.index("Email") + 1

# Track if customer was found

found = False

# Loop through rows to find matching Customer\_ID

for row in range(2, ws.max\_row + 1):

if ws.cell(row=row, column=id\_index).value == customer\_id:

ws.cell(row=row, column=email\_index).value = new\_email

found = True

break

if not found:

raise CustomerNotFoundException(f"Customer ID {customer\_id} not found.")

# Save the updated workbook

wb.save("updated\_customer.xlsx")

print(f"Email updated successfully for Customer ID {customer\_id}.")

except CustomerNotFoundException as e:

logging.error(str(e))

print(str(e))

except Exception as e:

logging.error("Unexpected error: " + str(e))

print("Unexpected error:", str(e))

# Example usage

update\_customer\_email("customer.xlsx", 104, "newmail@example.com")

5. Calculate Total Stock Value

Read product.xlsx, calculate the total stock value (Price \* Qty) for each product, and store the results in a new file stock\_value.xlsx. If a product has negative qty, raise a NegativeStockException.

from openpyxl import load\_workbook, Workbook

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Custom Exception

class NegativeStockException(Exception):

pass

# Function to calculate stock value

def calculate\_stock\_value(file\_path):

try:

# Load the product workbook

wb = load\_workbook(file\_path)

ws = wb["Products"] # assuming the sheet name is "Products"

# Get headers and column indexes

headers = [cell.value for cell in ws[1]]

id\_idx = headers.index("Product\_ID") + 1

name\_idx = headers.index("Product\_Name") + 1

price\_idx = headers.index("Price") + 1

qty\_idx = headers.index("Qty") + 1

# Create new workbook for output

new\_wb = Workbook()

new\_ws = new\_wb.active

new\_ws.title = "Stock\_Values"

# Write headers to new file

new\_ws.append(["Product\_ID", "Product\_Name", "Price", "Qty", "Stock\_Value"])

# Loop and compute stock value

for row in range(2, ws.max\_row + 1):

price = ws.cell(row=row, column=price\_idx).value

qty = ws.cell(row=row, column=qty\_idx).value

# Check for negative quantity

if qty < 0:

product\_id = ws.cell(row=row, column=id\_idx).value

raise NegativeStockException(f"Negative quantity found for Product ID: {product\_id}")

stock\_value = price \* qty

# Append to new workbook

new\_ws.append([

ws.cell(row=row, column=id\_idx).value,

ws.cell(row=row, column=name\_idx).value,

price,

qty,

stock\_value

])

# Save new workbook

new\_wb.save("stock\_value.xlsx")

print("Stock values calculated and saved in 'stock\_value.xlsx'.")

except NegativeStockException as e:

logging.error(str(e))

print("ERROR:", str(e))

except Exception as e:

logging.error("Unexpected error: " + str(e))

print("Unexpected error:", str(e))

# Example usage

calculate\_stock\_value("product.xlsx")

6. Merge Employee and Customer Data

Read employee.xlsx and customer.xlsx and merge records where employees are also customers (matching by email). Save the merged data to employee\_customer.xlsx. Handle missing email entries with EmailNotFoundException.

from openpyxl import load\_workbook, Workbook

import logging

# Setup logging

logging.basicConfig(filename="error\_log.txt", level=logging.ERROR)

# Custom Exception

class EmailNotFoundException(Exception):

pass

def merge\_employee\_customer(emp\_file, cust\_file):

try:

# Load workbooks

emp\_wb = load\_workbook(emp\_file)

cust\_wb = load\_workbook(cust\_file)

emp\_ws = emp\_wb["Employees"]

cust\_ws = cust\_wb["Customers"]

# Get employee headers

emp\_headers = [cell.value for cell in emp\_ws[1]]

emp\_email\_idx = emp\_headers.index("Email") + 1

# Get customer headers

cust\_headers = [cell.value for cell in cust\_ws[1]]

cust\_email\_idx = cust\_headers.index("Email") + 1

# Create dictionaries for fast lookup

customer\_data = {}

for row in cust\_ws.iter\_rows(min\_row=2, values\_only=True):

email = row[cust\_email\_idx - 1]

if not email:

raise EmailNotFoundException("Missing email in customer data.")

customer\_data[email] = row

# Create new workbook

merged\_wb = Workbook()

merged\_ws = merged\_wb.active

merged\_ws.title = "Merged\_Data"

# Combined headers

merged\_headers = emp\_headers + [h for h in cust\_headers if h not in emp\_headers]

merged\_ws.append(merged\_headers)

# Match employees to customers via email

for row in emp\_ws.iter\_rows(min\_row=2, values\_only=True):

email = row[emp\_email\_idx - 1]

if not email:

raise EmailNotFoundException("Missing email in employee data.")

if email in customer\_data:

cust\_row = customer\_data[email]

combined\_row = list(row) + [cust\_row[i] for i in range(len(cust\_headers)) if cust\_headers[i] not in emp\_headers]

merged\_ws.append(combined\_row)

# Save merged data

merged\_wb.save("employee\_customer.xlsx")

print("Merged employee-customer data saved in 'employee\_customer.xlsx'.")

except EmailNotFoundException as e:

logging.error(str(e))

print("ERROR:", str(e))

except Exception as e:

logging.error("Unexpected error: " + str(e))

print("Unexpected error:", str(e))

# Run function

merge\_employee\_customer("employee.xlsx", "customer.xlsx")

7. Generate a Sales Report from Multiple Sheets

Assume sales.xlsx contains multiple sheets with sales data. Write a Python program to read all sheets and generate a consolidated sales report. Handle the case where a sheet is missing using SheetNotFoundException.

from openpyxl import load\_workbook, Workbook

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Custom exception

class SheetNotFoundException(Exception):

pass

def generate\_sales\_report(file\_path):

try:

wb = load\_workbook(file\_path)

# Create a new workbook for consolidated data

report\_wb = Workbook()

report\_ws = report\_wb.active

report\_ws.title = "Consolidated\_Report"

report\_ws.append(["Sale\_ID", "Product", "Qty", "Price", "Sheet\_Name"])

for sheet in wb.sheetnames:

try:

ws = wb[sheet]

for row in ws.iter\_rows(min\_row=2, values\_only=True):

report\_ws.append(list(row) + [sheet])

except KeyError:

raise SheetNotFoundException(f"Sheet '{sheet}' not found in the workbook.")

report\_wb.save("consolidated\_sales\_report.xlsx")

print("Sales report generated successfully.")

except SheetNotFoundException as e:

logging.error(str(e))

print(str(e))

except Exception as e:

logging.error("Unexpected error: " + str(e))

print("Unexpected error:", str(e))

# Run function

generate\_sales\_report("sales.xlsx")

8. Delete Products with Zero Stock

Read product.xlsx and remove any product with zero qty. Save the updated file. If no such product exists, raise a NoZeroStockException.

from openpyxl import load\_workbook

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Custom exception for zero stock products

class NoZeroStockException(Exception):

pass

def delete\_zero\_stock\_products(file\_path):

try:

# Load workbook and sheet

wb = load\_workbook(file\_path)

ws = wb.active

# Identify header and column index for qty

headers = [cell.value for cell in ws[1]]

qty\_index = headers.index("Qty") + 1 # 1-based index

# List to hold rows to delete

rows\_to\_delete = []

# Check for products with zero stock and add to delete list

for row in ws.iter\_rows(min\_row=2, values\_only=True):

if row[qty\_index - 1] == 0:

rows\_to\_delete.append(row[0]) # Add product ID to delete list

# Remove rows with zero stock

if rows\_to\_delete:

for row in rows\_to\_delete:

for r in range(2, ws.max\_row + 1):

if ws.cell(row=r, column=1).value == row: # Product ID

ws.delete\_rows(r)

break

wb.save("updated\_product.xlsx")

print("Products with zero stock deleted and saved to 'updated\_product.xlsx'.")

else:

raise NoZeroStockException("No products with zero stock found.")

except NoZeroStockException as e:

logging.error(str(e))

print(str(e))

except Exception as e:

logging.error(f"Unexpected error: {str(e)}")

print("Unexpected error:", str(e))

# Run the function

delete\_zero\_stock\_products("product.xlsx")

9. Employee Bonus Calculation

Read employee.xlsx, calculate a 10% bonus for employees earning above $5000, and save the updated data to bonus.xlsx. If an employee already has a bonus column, raise a BonusAlreadyExistsException.

from openpyxl import load\_workbook

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Custom exception for bonus column already exists

class BonusAlreadyExistsException(Exception):

pass

def calculate\_employee\_bonus(file\_path):

try:

# Load workbook and sheet

wb = load\_workbook(file\_path)

ws = wb.active

# Check if bonus column already exists

if "Bonus" in [cell.value for cell in ws[1]]:

raise BonusAlreadyExistsException("Bonus column already exists.")

# Add bonus column header

ws.cell(row=1, column=ws.max\_column + 1).value = "Bonus"

# Calculate bonus for employees earning more than 5000

for row in ws.iter\_rows(min\_row=2):

salary = row[4].value # Assuming salary is in the 5th column

if salary > 5000:

bonus = salary \* 0.10 # 10% bonus

ws.cell(row=row[0].row, column=ws.max\_column).value = bonus

# Save the updated workbook

wb.save("bonus.xlsx")

print("Employee bonus calculated and saved to 'bonus.xlsx'.")

except BonusAlreadyExistsException as e:

logging.error(str(e))

print(str(e))

except Exception as e:

logging.error(f"Unexpected error: {str(e)}")

print("Unexpected error:", str(e))

# Run the function

calculate\_employee\_bonus("employee.xlsx")

10. Backup and Restore Customer Data

Create a backup of customer.xlsx as customer\_backup.xlsx. Allow restoring data from the backup when requested. If the backup file is missing, raise a BackupNotFoundException.

import shutil

import logging

# Setup logging

logging.basicConfig(filename='error\_log.txt', level=logging.ERROR)

# Custom exception for backup file not found

class BackupNotFoundException(Exception):

pass

def backup\_and\_restore\_customer\_data(action="backup"):

try:

if action == "backup":

shutil.copy("customer.xlsx", "customer\_backup.xlsx")

print("Backup created as 'customer\_backup.xlsx'.")

elif action == "restore":

try:

shutil.copy("customer\_backup.xlsx", "customer.xlsx")

print("Data restored from 'customer\_backup.xlsx'.")

except FileNotFoundError:

raise BackupNotFoundException("Backup file 'customer\_backup.xlsx' not found.")

else:

raise ValueError("Invalid action. Use 'backup' or 'restore'.")

except BackupNotFoundException as e:

logging.error(str(e))

print(str(e))

except Exception as e:

logging.error(f"Unexpected error: {str(e)}")

print("Unexpected error:", str(e))

# Run the function (backup or restore)

backup\_and\_restore\_customer\_data("backup") # or "restore"