oops

December 22, 2024

0.1 Write a class called Product. The class should have fields called name, amount, and price, holding the product's name, the number of items of that product in stock, and the regular price of the product. There should be a method get_price that receives the number of items to be bought and returns a the cost of buying that many items, where the regular price is charged for orders of less than 10 items, a 10% discount is applied for orders of between 10 and 99 items, and a 20% discount is applied for orders of 100 or more items. There should also be a method called make_purchase that receives the number of items to be bought and decreases amount by that much.

[1]:
```
class Product:

    def __init__(self, name, amount, price):
        self.name = name
        self.amount = amount
        self.price = price

    def get_price(self, number_to_be_bought):
        discount = 0
        if number_to_be_bought < 10:
            pass
        elif 10 <= number_to_be_bought < 99:
            discount = 10
        else:
            discount = 20
        price = (100 - discount) / 100 * self.price
        return price * number_to_be_bought

    def make_purchase(self, quantity):
        self.amount -= quantity


# name = input('name:')
# amount = int(input('Digit amount of items'))
# price = int(input('Digit price of items'))

name, amount, price = 'shoes', 200, 33
```

```
shoes = Product(name, amount, price)
# quantity = int(input('Digit amount of items to buy'))

q1 = 4
print(f'cost for {q1} {shoes.name} = {shoes.get_price(q1)}')
shoes.make_purchase(q1)
print(f'remaining stock: {shoes.amount}\n')

q2 = 12
print(f'cost for {q2} {shoes.name} = {shoes.get_price(q2)}')
shoes.make_purchase(q2)
print(f'remaining stock: {shoes.amount}\n')

q3 = 112
print(f'cost for {q3} {shoes.name} = {shoes.get_price(q3)}')
shoes.make_purchase(q3)
print(f'remaining stock: {shoes.amount}\n')
```

```
cost for 4 shoes = 132.0
remaining stock: 196

cost for 12 shoes = 356.4
remaining stock: 184

cost for 112 shoes = 2956.8
remaining stock: 72
```

0.2 You need to create the foundations of an e-commerce engine for a B2C (business-to-consumer) retailer. You need to have a class for a customer called User, a class for items in inventory called Item, and a shopping cart class called Cart. Items go in Carts, and Users can have multiple Carts. Also, multiple items can go into Carts, including more than one of any single item.

[2]:
```
class User:
    def __init__(self, id, name):
        self.id = id
        self.name = name
    def display_user(self):
        print('ID', self.id, 'Name:', self.name)

class Item:
    def __init__(self, id, name, price, sold, available):
        self.id = id
        self.name = name
```

```
        self.price = price
        self.sold = sold
        self.available = available

class Cart:
    def __init__(self, user):
        self.user = user
        self.cart_items = []
    def insert_items(self, item, quantity):
        for i in range(quantity):
            if item.available == 0:
                print('Out of stock')
                break
            self.cart_items.append(item)
            item.sold += 1
            item.available -= 1
    def display_cart(self):
        print('This Cart belongs to', self.user.name, 'with ID', self.user.id)
        self.total = 0
        for i in self.cart_items:
            print('Item',i.name)
            self.total += i.price
        print('Total price = ', self.total)



#Creating a user
user1 = User(1, 'Kavit')
user1.display_user()

#Creating items
apple = Item(1, 'apple', 100, 0, 10)
greencoconut = Item(2, 'green coconut', 150, 0, 3)
milk = Item(2, 'milk', 24, 0, 100)

#Creating two carts for user1
cart1 = Cart(user1)
cart2 = Cart(user1)

#Adding items to cart1
cart1.insert_items(apple, 2)
cart1.insert_items(milk, 3)

#Adding items to cart2
cart2.insert_items(greencoconut, 3)
cart2.insert_items(milk, 20)
```

```
#Displaying details of cart1 and cart2
cart1.display_cart()
cart2.display_cart()
```

```
ID 1 Name: Kavit
This Cart belongs to Kavit with ID 1
Item apple
Item apple
Item milk
Item milk
Item milk
Total price =  272
This Cart belongs to Kavit with ID 1
Item green coconut
Item green coconut
Item green coconut
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Item milk
Total price =  930
```

[1]:
```
class Pizza:
    def __init__(self, size, toppings, cheese):
        self.size = size
        self.toppings = toppings
        self.cheese = cheese

    def price(self):
        self.cost = 0
        if self.size == 'small':
```

```
            self.cost += 50
        elif self.size == 'medium':
            self.cost += 100
        else:
            self.cost += 200
        topping_prices_20 = ['corn', 'tomato', 'onion', 'capsicum']
        topping_prices_50 = ['mushroom', 'olives', 'broccoli']
        for topping in self.toppings:
            if topping in topping_prices_20:
                self.cost += 20
            else:
                self.cost += 50
        #for cheese
        self.cost += 50 * len(self.cheese)
        return self.cost

class Order:
    def __init__(self, name, customerid):
        self.name = name
        self.customerid = customerid

    def order(self, n):
        self.pizzas = []
        for i in range(n):
            toppings = []
            cheese = []
            print('Customize Pizza',i+1)
            size = input('Select size: ')
            t = int(input('How many toppings: '))
            for i in range(t):
                toppings.append(input('Enter toppings: '))
            t = int(input('How many cheese: '))
            for i in range(t):
                cheese.append(input('Enter cheese: '))
            self.pizzas.append(Pizza(size, toppings, cheese))

    def bill(self):
        self.total = 0
        count = 1
        for p in self.pizzas:
            print('Pizza', count)
            print(p.size, p.toppings, p.cheese)
            self.total += p.price()
            count += 1
        print('Total bill amount:', self.total)

number=int(input("How many pizzas you want to order: "))
```

```
order1 = Order('Kavit', 1)
order1.order(number)
order1.bill()
```

```
How many pizzas you want to order: 2
Customize Pizza 1
Select size: small
How many toppings: 1
Enter toppings: corn
How many cheese: 0
Customize Pizza 2
Select size: small
How many toppings: 0
How many cheese: 0
Pizza 1
small ['corn'] []
Pizza 2
small [] []
Total bill amount: 120
```

0.3 Write a program to build a simple Student Management System using Python which can perform the following operations:

1. Accept
2. Display
3. Search
4. Delete
5. Update

0.3.1 Accept – This method takes details from the user like name, roll number, and marks for two different subjects.

0.3.2 Display – This method displays the details of every student.

0.3.3 Search – This method searches for a particular student from the list of students. This method will ask the user for roll number and then search according to the roll number

0.3.4 Delete – This method deletes the record of a particular student with a matching roll number.

0.3.5 Update – This method updates the roll number of the student. This method will ask for the old roll number and new roll number. It will replace the old roll number with a new roll number.

[ ]:
```
# This is simplest Student data management program in python

# Create class "Student"
class Student:
```

```
# Constructor
    def __init__(self, name, rollno, m1, m2):
        self.name = name
        self.rollno = rollno
        self.m1 = m1
        self.m2 = m2

    # Function to create and append new student
    def accept(self):
        Name=input("Enter Name: ")
        Rollno=int(input("Enter Rollno.: "))
        marks1=int(input("Enter marks of 1: "))
        marks2=int(input("Enter marks of 2: "))
        ob = Student(Name, Rollno, marks1, marks2)
        ls.append(ob)

    # Function to display student details
    def display(self, ob):
        print("Name : ", ob.name)
        print("RollNo : ", ob.rollno)
        print("Marks1 : ", ob.m1)
        print("Marks2 : ", ob.m2)
        print("\n")

    # Search Function
    def search(self, rn):
        for i in range(ls.__len__()):
            if(ls[i].rollno == rn):
                return i

    # Delete Function
    def delete(self, rn):
        i = obj.search(rn)
        del ls[i]

    # Update Function
    def update(self, rn, No):
        i = obj.search(rn)
        roll = No
        ls[i].rollno = roll


# Create a list to add Students
ls = []
# an object of Student class
obj = Student('', 0, 0, 0)
```

```
print("\nOperations used, ")
print("\n1.Accept Student details\n2.Display Student Details\n3.Search Details␣
 ↪of a Student\n4.Delete Details of Student\n5.Update Student Details\n6.Exit")

# ch = int(input("Enter choice:"))
# if(ch == 1):
obj.accept()
obj.accept()
obj.accept()

# elif(ch == 2):
print("\n")
print("\nList of Students\n")
for i in range(ls.__len__()):
    obj.display(ls[i])

# elif(ch == 3):
print("\n Student Found, ")
s = obj.search(10)
obj.display(ls[s])

# elif(ch == 4):
obj.delete(30)
print(ls.__len__())
print("List after deletion")
for i in range(ls.__len__()):
    obj.display(ls[i])

# elif(ch == 5):
obj.update(20,15)
print(ls.__len__())
print("List after updation")
for i in range(ls.__len__()):
    obj.display(ls[i])

# else:
print("Thank You !")
```

0.4 Stacks and Queues. Write a class which defines a data structure that can behave as both a queue (FIFO) or a stack (LIFO), There are four methods that should be implemented:

0.4.1 shift() returns the first element and removes it from the list

0.4.2 unshift() "pushes" a new element to the front or head of the list

0.4.3 push() adds a new element to the end of a list

0.4.4 pop() returns the last element and removes it from the list

[ ]:
```
class StackQueue:
    def __init__(self, L):
        self.L = L
    def shift(self):
        if len(self.L) == 0:
            pass
            raise Exception('List is empty')
        try:
            x = self.L.pop(0)
            return x
        except Exception:
            print(Exception)
    def unshift(self, n):
        self.L.insert(0, n)
    def push(self, n):
        self.L.append(n)
    def pop(self):
        if len(self.L) == 0:
            raise Exception('List is empty')
        try:
            x = self.L.pop()
            return x
        except Exception:
            print(Exception)
    def display(self):
        return self.L

sq = StackQueue([1,4,6,8,9])
print(sq.shift())
sq.unshift(7)
sq.push(10)
print(sq.pop())
print(sq.display())
```

1 Write the definition of a Point class. Objects from this class should have a

a method show to display the coordinates of the point

a method move to change these coordinates.

a method dist that computes the distance between 2 points.

[ ]:
```
import math


class Point(object):
    """Class to handle point in a 2 dimensions space"""

    def __init__(self, x, y):
        """
        :param x: the value on the X-axis
        :type x: float
        :param y: the value on the Y-axis
        :type y: float
        """
        self.x = x
        self.y = y


    def show(self):
        """
        :return: the coordinate of this point
        :rtype: a tuple of 2 elements (float, float)
        """
        return self.x, self.y


    def move(self, x, y):
        """
        :param x: the value to move on the X-axis
        :type x: float
        :param y: the value to move on the Y-axis
        :type y: float
        """
        self.x += x
        self.y += y


    def dist(self, pt):
        """
        :param pt: the point to compute the distance with
```

```
        :type pt: :class:`Point` object
        :return: the distance between this point ant pt
        :rtype: int
        """
        dx = pt.x - self.x
        dy = pt.y - self.y
        return math.sqrt(dx ** 2 + dy ** 2)
```

[ ]:
```
p1 = Point(2, 3)
p2 = Point(3, 3)
print(p1.show())
print(p2.show())
p1.move(10, -10)
print(p1.show())
print(p2.show())
print(p1.dist(p2))
```

2 Get index in the list of objects by attribute in Python

[ ]:
```
class X:
    def __init__(self,val):
        self.val = val

def getIndex(li,target):
    for index, x in enumerate(li):
        if x.val == target:
            return index
    return -1

# Driver code
li = [1,2,3,4,5,6]

# Converting all the items in
# list to object of class X
a = list()
for i in li:
    a.append(X(i))

print(getIndex(a,3))
```

3 How to create a list of object in Python class

[ ]:
```
class geeks:
    def __init__(self, name, roll):
        self.name = name
```

```
        self.roll = roll

# creating list
list = []

# appending instances to list
list.append(geeks('Akash', 2))
list.append(geeks('Deependra', 40))
list.append(geeks('Reaper', 44))
list.append(geeks('veer', 67))

# Accessing object value using a for loop
for obj in list:
    print(obj.name, obj.roll, sep=' ')

print("")
# Accessing individual elements
print(list[0].name)
print(list[1].name)
print(list[2].name)
print(list[3].name)
```

4 Create a Sphere class that accepts a radius upon instantiation and has a volume and surface area method.

[ ]:
```
class Sphere():

    def __init__(self,radius):
        self.radius = radius

    def volume(self):
        return (4/3)*3.14*self.radius**3

    def surface_area(self):
        return 4 *3.14 * self.radius **2
```

[ ]:
```
s = Sphere(1)
print(s.surface_area())
print(s.volume())
```

Write a Rectangle class in Python language, allowing you to build a rectangle with length and width attributes.

Create a Perimeter() method to calculate the perimeter of the rectangle and a Area() method to calculate the area of the rectangle.

Create a method display() that display the length, width, perimeter and area of an object created using an instantiation on rectangle class.

[ ]:
```python
class Rectangle:
    # define constructor with attributes: length and width
    def __init__(self, length , width):
        self.length = length
        self.width = width

    # Create Perimeter method
    def Perimeter(self):
        return 2*(self.length + self.width)

    # Create area method
    def Area(self):
        return self.length*self.width

    # create display method
    def display(self):
        print("The length of rectangle is: ", self.length)
        print("The width of rectangle is: ", self.width)
        print("The perimeter of rectangle is: ", self.Perimeter())
        print("The area of rectangle is: ", self.Area())
```

[ ]:
```python
myRectangle = Rectangle(7 , 5)
myRectangle.display()
print("-----------------------------------")
```

5 Create a Python class called BankAccount which represents a bank account, having as attributes: accountNumber (numeric type), name (name of the account owner as string type), balance.

Create a constructor with parameters: accountNumber, name, balance.

Create a Deposit() method which manages the deposit actions.

Create a Withdrawal() method which manages withdrawals actions.

Create an bankFees() method to apply the bank fees with a percentage of 5% of the balance account.

Create a display() method to display account details.

Give the complete code for the BankAccount class.

[ ]:
```python
class BankAccount:
    # create the constuctor with parameters: accountNumber, name and balance
    def __init__(self,accountNumber, name, balance):
        self.accountNumber = accountNumber
        self.name = name
        self.balance = balance
```

```
    # create Deposit() method
    def Deposit(self , d ):
        self.balance = self.balance + d

    # create Withdrawal method
    def Withdrawal(self , w):
        if(self.balance < w):
            print("impossible operation! Insufficient balance !")
        else:
            self.balance = self.balance - w
    # create bankFees() method
    def bankFees(self):
        self.balance = (95/100)*self.balance

    # create display() method
    def display(self):
        print("Account Number : " , self.accountNumber)
        print("Account Name : " , self.name)
        print("Account Balance : " , self.balance , " $")

# Testing the code :
newAccount = BankAccount(2178514584, "Albert" , 2700)
# Creating Withdrawal Test
newAccount.Withdrawal(300)
# Create deposit test
newAccount.Deposit(200)
# Display account informations
newAccount.display()
```

6 1 - Create a Coputation class with a default constructor (without parameters) allowing to perform various calculations on integers numbers.

2 - Create a method called Factorial() which allows to calculate the factorial of an integer. Test the method by instantiating the class.

3 - Create a method called Sum() allowing to calculate the sum of the first n integers 1 + 2 + 3 + .. + n. Test this method.

4 - Create a method called testPrim() in the Calculation class to test the primality of a given integer. Test this method.

4 - Create a method called testPrims() allowing to test if two numbers are prime between them.

5 - Create a tableMult() method which creates and displays the multiplication table of a given integer. Then create an allTablesMult() method to display all the integer multiplication tables 1, 2, 3, ..., 9.

6 - Create a static listDiv() method that gets all the divisors of a given integer on new list called Ldiv. Create another listDivPrim() method that gets all the prime divisors of a given integer.

[ ]:
```
class Computation:
    def __init__ (self):
        pass
# --- Factorial ------------
    def factorial(self, n):
        j = 1
        for i in range (1, n + 1):
            j = j * i
        return j

# --- Sum of the first n numbers ----
    def sum (self, n):
        j = 1
        for i in range (1, n + 1):
            j = j + i
        return j

# --- Primality test of a number ------------
    def testPrim (self, n):
        j = 0
        for i in range (1, n + 1):
            if (n% i == 0):
                j = j + 1
        if (j == 2):
            return True
        else:
            return False

# --- Primality test of two integers ------------
    def testprims (self, n, m):

        # initialize the number of commons divisors
        commonDiv = 0
        for i in range (1, n + 1):
            if (n% i == 0 and m% i == 0):
                commonDiv = commonDiv + 1
        if commonDiv == 1:
            print ("The numbers", n, "and", m, "are co-primes")
        else:
            print ("The numbers", n, "and", m, "are not co-primes")

#---Multiplication table-------------
    def tableMult (self, k):
        for i in range (1,10):
```

```
            print (i, "x", k, "=", i * k)

# --- All multiplication tables of the numbers 1, 2, .., 9
    def allTables (self):
        for k in range (1,10):
            print ("\nthe multiplication table of:", k, "is:")
            for i in range (1,10):
                print (i, "x", k, "=", i * k)

# ----- list of divisors of an integer
    def listDiv (self, n):
        # initialization of the list of divisors
        lDiv = []
        for i in range (1, n + 1):
            if (n% i == 0):
                lDiv.append (i)
        return lDiv

# ------ list of prime divisors of an integer ----------------
    def listDivPrim (self, n):
        # initialization of the list of divisors
        lDiv = []
        for i in range (1, n + 1):
            if (n% i == 0 and self.testPrim (i)):
                lDiv.append (i)
        return lDiv

# Instantiation example
Comput= Computation ()
Comput.testprims (13, 7)
print ("List of divisors of 18:", Comput.listDiv (18))
print ("List of prime divisors of 18:", Comput.listDivPrim (18))
Comput.allTables ()
```

[1]:
```
#write a python program that has class store which keeps record of code and␣
 ↪price of
#each product. Display a menu of all products to the user and prompt
#him to enter the quantity of each item required . generate a bill and display␣
 ↪total amount.
class Store:
    def __init__(self,n):
        self.item_code=[]
        self.price=[]
        self.n=n
    def get_data(self):
        #n=int(input("enter no of items: "))
        for i in range(self.n):
```

```
            self.item_code.append(int(input("enter code of item: ")))
            self.price.append(int(input("enter cost of item: ")))
    def display_data(self):
        print("Item Code \t Price")
        for i in range(self.n):
            print(self.item_code[i],"\t\t ",self.price[i])

    def calculate_bill(self,quant):
        total_amount=0
        for i in range(self.n):
            total_amount+=(self.price[i]*quant[i])
        print("************************Bill**********************")
        print("ITEM \t PRICE \t QUANTITY \t SUBTOTAL")
        for i in range(self.n):
            print(self.item_code[i],"\t",self.price[i],"\t",quant[i],"\t\t",
                  self.price[i]*quant[i])
        print("************************************")
        print("Total= ",total_amount)

n=int(input("enter no of items: "))
s1=Store(n)
s1.get_data()
s1.display_data()
q=[]
print("Enter quantity of each item: ")
for i in range(n):
    q.append(int(input("Enter quantity of item {} : ".format(i+1))))

s1.calculate_bill(q)
```

```
enter no of items: 1
enter code of item: 1
enter cost of item: 25
Item Code        Price
1                 25
Enter quantity of each item:
Enter quantity of item 1 : 5
************************Bill**********************
ITEM     PRICE   QUANTITY        SUBTOTAL
1        25      5               125
************************************
Total=  125
```

[2]:
```
#write a python program for library book record with oops.

class library:
```

```
    def __init__(self):
        self.title=""
        self.author=""
        self.publisher=""
    def read(self):
        self.title=input("Enter Book Title: ")
        self.author=input("Enter Book author: ")
        self.publisher=input("Enter Book Publisher: ")
    def display(self):
        print("Title:", self.title)
        print("Author:", self.author)
        print("Publisher:", self.publisher)
        print("\n")
my_book=[]
ch='y'
while(ch=='y'):
    print('''
1. Add New Book
2. Display Books
''')
    choice=int(input("Enter choice: "))
    if(choice==1):
        book=library()
        book.read()
        my_book.append(book)
    elif(choice==2):
        for i in my_book:
            i.display()
    else:
        print("Invalid choice!")
    ch=input("Do you want to continue..?")
print("Bye!")
```

```
1. Add New Book
2. Display Books

Enter choice: 1
Enter Book Title: vha
Enter Book author: vishal
Enter Book Publisher: lj
Do you want to continue..?2
Bye!
```

Problem Definition: Habit Tracker Application for Students

- Problem Statement:

Students often struggle with managing their daily habits, such as studying, exercise, or other self-improvement activities. A habit tracker application can help students set goals, monitor their progress, and stay motivated. The problem is to design a system that allows students to track their habits, set target streaks, update their progress, and analyze their overall performance.

- Objective:

1. Design a Habit Tracker application for students that will allow them to:

2. Add Habits: Students can set habits they wish to track (e.g., study, exercise, reading).

3. Set Target Streak: For each habit, students can define a target streak (e.g., 30 days of continuous exercise).

- Track Progress: The system should track the student's progress in each habit by updating the streak each time the habit is completed.
- Analyze Progress: Calculate the overall progress for all habits and display a summary for the student.
- Reset or Update Streak: The system should allow students to reset or update their streak if necessary.

Features:

The application should implement the following features:

1. Habit Class:

- A class to represent a habit with the following attributes:

1. name: Name of the habit (e.g., "Exercise").

2. target_streak: The number of days the student aims to complete the habit consecutively (e.g., 30 days).
3. current_streak: The current number of consecutive days the habit has been completed.

- Methods in the Habit class:

1. update_streak(): Updates the current streak by 1 day.
2. reset_streak(): Resets the current streak to 0 days.
3. progress(): Returns the percentage progress towards the target streak.
4. HabitTracker Class:

A class to manage multiple habits, allowing students to: Add a Habit: Add a new habit to the tracker with a specified target streak.

- Display Habits: Display all habits along with their current streak and progress.
- Update a Habit: Update the streak of a specific habit when completed.
- Reset a Habit: Reset the streak of a specific habit to 0 days.
- Calculate Overall Progress: Calculate and display the overall progress across all tracked habits.

Find the Habit with the Highest Streak: Display which habit has the highest current streak.

- Input:

A list of habits with names and target streaks. Operations to update, reset, or analyze the habits.

- Output:

A summary of habits, showing the name, target streak, current streak, and progress. Overall progress percentage for all habits. The habit with the highest streak. Updated progress after actions like resetting or updating a habit.

- Constraints:

The target streak for each habit will be a positive integer (e.g., 1-365 days). The current streak will be a non-negative integer (initially 0). The habit tracker should handle multiple habits and allow students to track up to hundreds of habits simultaneously. Example Use Case:

- Add Habits:

The student adds three habits: "Exercise" (target streak: 30 days), "Reading" (target streak: 15 days), "Meditation" (target streak: 18 days).

- Update Streak:

The student updates their streak after completing their habits. After updating, the current streak for "Exercise" becomes 2 days, for "Reading" 1 day, and for "Meditation" 1 day.

- Display Progress:

The student views their progress for all habits, which shows the percentage of completion relative to the target streak for each habit.

- Reset Streak:

The student resets their streak for "Reading," and the tracker reflects this change, updating the progress for that habit. Calculate Overall Progress:

The system calculates and shows the overall progress, averaging the progress of all tracked habits. Find Habit with Highest Streak:

The system identifies the habit with the longest streak, showing that "Exercise" has the highest current streak.

In [1]:
```
class Habit:
    def __init__(self, name, target_streak):
        self.name = name
        self.target_streak = target_streak
        self.current_streak = 0

    def update_streak(self):
        self.current_streak += 1

    def reset_streak(self):
```

```
        self.current_streak = 0

    def progress(self):
        return (self.current_streak / self.target_streak) * 100


class HabitTracker:
    def __init__(self):
        self.habits = []

    def add_habit(self, name, target_streak):
        habit = Habit(name, target_streak)
        self.habits.append(habit)
        print(f"Habit '{name}' added to tracker.")

    def display_habits(self):
        if not self.habits:
            print("No habits being tracked.")
            return

        print("\nHabit Tracker:")
        for habit in self.habits:
            progress = habit.progress()
            print(f"Habit: {habit.name}, Target Streak: {habit.target_
                  f"Current Streak: {habit.current_streak} days, Progr

    def update_streak(self, habit_name):
        for habit in self.habits:
            if habit.name == habit_name:
                habit.update_streak()
                print(f"'{habit_name}' streak updated! Current streak:
                return
        print(f"Habit '{habit_name}' not found.")

    def reset_streak(self, habit_name):
        for habit in self.habits:
            if habit.name == habit_name:
                habit.reset_streak()
                print(f"'{habit_name}' streak has been reset.")
                return
        print(f"Habit '{habit_name}' not found.")

    def calculate_overall_progress(self):
        if not self.habits:
            print("No habits being tracked.")
            return
```

```
        total_progress = sum(habit.progress() for habit in self.habits
        overall_progress = total_progress / len(self.habits)
        print(f"\nOverall Progress Across All Habits: {overall_progres

    def find_highest_streak(self):
        if not self.habits:
            print("No habits being tracked.")
            return

        highest_streak_habit = max(self.habits, key=lambda habit: habi
        print(f"\nHabit with the Highest Streak: {highest_streak_habit
              f"with {highest_streak_habit.current_streak} days.")

# Instantiate the HabitTracker
tracker = HabitTracker()

# Adding habits
tracker.add_habit("Exercise", 30)
tracker.add_habit("Reading", 15)
tracker.add_habit("Meditation", 18)

# Displaying all habits
tracker.display_habits()

# Updating streaks
tracker.update_streak("Exercise")
tracker.update_streak("Reading")
tracker.update_streak("Meditation")
tracker.update_streak("Exercise")  # Updating streak again

# Displaying updated habits
tracker.display_habits()

# Calculating overall progress
tracker.calculate_overall_progress()

# Finding the habit with the highest streak
tracker.find_highest_streak()

# Resetting a habit's streak
tracker.reset_streak("Reading")

# Displaying habits after reset
tracker.display_habits()

# Recalculating overall progress
tracker.calculate_overall_progress()
```

```
Habit 'Exercise' added to tracker.
Habit 'Reading' added to tracker.
Habit 'Meditation' added to tracker.

Habit Tracker:
Habit: Exercise, Target Streak: 30 days, Current Streak: 0 days, Progre
ss: 0.00%
Habit: Reading, Target Streak: 15 days, Current Streak: 0 days, Progres
s: 0.00%
Habit: Meditation, Target Streak: 18 days, Current Streak: 0 days, Prog
ress: 0.00%
'Exercise' streak updated! Current streak: 1
'Reading' streak updated! Current streak: 1
'Meditation' streak updated! Current streak: 1
'Exercise' streak updated! Current streak: 2

Habit Tracker:
Habit: Exercise, Target Streak: 30 days, Current Streak: 2 days, Progre
ss: 6.67%
Habit: Reading, Target Streak: 15 days, Current Streak: 1 days, Progres
s: 6.67%
Habit: Meditation, Target Streak: 18 days, Current Streak: 1 days, Prog
ress: 5.56%

Overall Progress Across All Habits: 6.30%

Habit with the Highest Streak: Exercise with 2 days.
'Reading' streak has been reset.

Habit Tracker:
Habit: Exercise, Target Streak: 30 days, Current Streak: 2 days, Progre
ss: 6.67%
Habit: Reading, Target Streak: 15 days, Current Streak: 0 days, Progres
s: 0.00%
Habit: Meditation, Target Streak: 18 days, Current Streak: 1 days, Prog
ress: 5.56%

Overall Progress Across All Habits: 4.07%
```


In [ ]:

file

December 26, 2024

[4]:
```
#find japanese word in text file
import string
file=open("secretmessage.txt",encoding="utf-8")
japanese=""
for line in file:
    for i in line:
        if (i.lower() in ("abcdefghijklmnopqrstuvwxyz")) or (i in string.
 ↪punctuation) or i in (" ","\n"):
            continue
        japanese+=i
print(japanese)
```

[5]:
```
#meaning of this word is first char of each line
file=open("secretmessage.txt",encoding="utf-8")
meaning=""
for line in file:
    meaning+=line[0]
print(meaning)
```

```
EVENAMONKEYFALLSFROMATREE
```

oop_program_vha

December 26, 2024

1 Problem Definition: Habit Tracker Application for Students

- Problem Statement: Students often struggle with managing their daily habits, such as studying, exercise, or other self-improvement activities. A habit tracker application can help students set goals, monitor their progress, and stay motivated. The problem is to design a system that allows students to track their habits, set target streaks, update their progress, and analyze their overall performance.
- Objective:

1. Design a Habit Tracker application for students that will allow them to:
2. Add Habits: Students can set habits they wish to track (e.g., study, exercise, reading).
3. Set Target Streak: For each habit, students can define a target streak (e.g., 30 days of continuous exercise).

- Track Progress: The system should track the student's progress in each habit by updating the streak each time the habit is completed.
- Analyze Progress: Calculate the overall progress for all habits and display a summary for the student.
- Reset or Update Streak: The system should allow students to reset or update their streak if necessary. ###### Features: The application should implement the following features:

1. Habit Class:

- A class to represent a habit with the following attributes:

1. name: Name of the habit (e.g., "Exercise").
2. target_streak: The number of days the student aims to complete the habit consecutively (e.g., 30 days).
3. current_streak: The current number of consecutive days the habit has been completed.

- Methods in the Habit class:

1. update_streak(): Updates the current streak by 1 day.
2. reset_streak(): Resets the current streak to 0 days.
3. progress(): Returns the percentage progress towards the target streak.
4. HabitTracker Class:

A class to manage multiple habits, allowing students to: Add a Habit: Add a new habit to the tracker with a specified target streak. - Display Habits: Display all habits along with their current

streak and progress. - Update a Habit: Update the streak of a specific habit when completed. - Reset a Habit: Reset the streak of a specific habit to 0 days. - Calculate Overall Progress: Calculate and display the overall progress across all tracked habits.

Find the Habit with the Highest Streak: Display which habit has the highest current streak. - Input: A list of habits with names and target streaks. Operations to update, reset, or analyze the habits. - Output: A summary of habits, showing the name, target streak, current streak, and progress. Overall progress percentage for all habits. The habit with the highest streak. Updated progress after actions like resetting or updating a habit. - Constraints: The target streak for each habit will be a positive integer (e.g., 1-365 days). The current streak will be a non-negative integer (initially 0). The habit tracker should handle multiple habits and allow students to track up to hundreds of habits simultaneously. Example Use Case: - Add Habits:

The student adds three habits: “Exercise” (target streak: 30 days), “Reading” (target streak: 15 days), “Meditation” (target streak: 18 days). - Update Streak:

The student updates their streak after completing their habits. After updating, the current streak for “Exercise” becomes 2 days, for “Reading” 1 day, and for “Meditation” 1 day. - Display Progress:

The student views their progress for all habits, which shows the percentage of completion relative to the target streak for each habit. - Reset Streak:

The student resets their streak for “Reading,” and the tracker reflects this change, updating the progress for that habit. Calculate Overall Progress:

The system calculates and shows the overall progress, averaging the progress of all tracked habits. Find Habit with Highest Streak:

The system identifies the habit with the longest streak, showing that “Exercise” has the highest current streak.

[1]:
```
class Habit:
    def __init__(self, name, target_streak):
        self.name = name
        self.target_streak = target_streak
        self.current_streak = 0

    def update_streak(self):
        self.current_streak += 1

    def reset_streak(self):
        self.current_streak = 0

    def progress(self):
        return (self.current_streak / self.target_streak) * 100


class HabitTracker:
    def __init__(self):
        self.habits = []
```

```
    def add_habit(self, name, target_streak):
        habit = Habit(name, target_streak)
        self.habits.append(habit)
        print(f"Habit '{name}' added to tracker.")

    def display_habits(self):
        if not self.habits:
            print("No habits being tracked.")
            return

        print("\nHabit Tracker:")
        for habit in self.habits:
            progress = habit.progress()
            print(f"Habit: {habit.name}, Target Streak: {habit.target_streak}␣
↪days, "
                  f"Current Streak: {habit.current_streak} days, Progress:␣
↪{progress:.2f}%")

    def update_streak(self, habit_name):
        for habit in self.habits:
            if habit.name == habit_name:
                habit.update_streak()
                print(f"'{habit_name}' streak updated! Current streak: {habit.
↪current_streak}")
                return
        print(f"Habit '{habit_name}' not found.")

    def reset_streak(self, habit_name):
        for habit in self.habits:
            if habit.name == habit_name:
                habit.reset_streak()
                print(f"'{habit_name}' streak has been reset.")
                return
        print(f"Habit '{habit_name}' not found.")

    def calculate_overall_progress(self):
        if not self.habits:
            print("No habits being tracked.")
            return

        total_progress = sum(habit.progress() for habit in self.habits)
        overall_progress = total_progress / len(self.habits)
        print(f"\nOverall Progress Across All Habits: {overall_progress:.2f}%")

    def find_highest_streak(self):
        if not self.habits:
            print("No habits being tracked.")
```

```
            return

        highest_streak_habit = max(self.habits, key=lambda habit: habit.
 ↪current_streak)
        print(f"\nHabit with the Highest Streak: {highest_streak_habit.name} "
              f"with {highest_streak_habit.current_streak} days.")

# Instantiate the HabitTracker
tracker = HabitTracker()

# Adding habits
tracker.add_habit("Exercise", 30)
tracker.add_habit("Reading", 15)
tracker.add_habit("Meditation", 18)

# Displaying all habits
tracker.display_habits()

# Updating streaks
tracker.update_streak("Exercise")
tracker.update_streak("Reading")
tracker.update_streak("Meditation")
tracker.update_streak("Exercise")  # Updating streak again

# Displaying updated habits
tracker.display_habits()

# Calculating overall progress
tracker.calculate_overall_progress()

# Finding the habit with the highest streak
tracker.find_highest_streak()

# Resetting a habit's streak
tracker.reset_streak("Reading")

# Displaying habits after reset
tracker.display_habits()

# Recalculating overall progress
tracker.calculate_overall_progress()
```

```
Habit 'Exercise' added to tracker.
Habit 'Reading' added to tracker.
Habit 'Meditation' added to tracker.

Habit Tracker:
Habit: Exercise, Target Streak: 30 days, Current Streak: 0 days, Progress: 0.00%
```

```
Habit: Reading, Target Streak: 15 days, Current Streak: 0 days, Progress: 0.00%
Habit: Meditation, Target Streak: 18 days, Current Streak: 0 days, Progress:
0.00%
'Exercise' streak updated! Current streak: 1
'Reading' streak updated! Current streak: 1
'Meditation' streak updated! Current streak: 1
'Exercise' streak updated! Current streak: 2

Habit Tracker:
Habit: Exercise, Target Streak: 30 days, Current Streak: 2 days, Progress: 6.67%
Habit: Reading, Target Streak: 15 days, Current Streak: 1 days, Progress: 6.67%
Habit: Meditation, Target Streak: 18 days, Current Streak: 1 days, Progress:
5.56%

Overall Progress Across All Habits: 6.30%

Habit with the Highest Streak: Exercise with 2 days.
'Reading' streak has been reset.

Habit Tracker:
Habit: Exercise, Target Streak: 30 days, Current Streak: 2 days, Progress: 6.67%
Habit: Reading, Target Streak: 15 days, Current Streak: 0 days, Progress: 0.00%
Habit: Meditation, Target Streak: 18 days, Current Streak: 1 days, Progress:
5.56%

Overall Progress Across All Habits: 4.07%
```

[ ]: