No.1

class Clock:

    def \_\_init\_\_(self, hour, minute, second):

        self.hour = hour

        self.minute = minute

        self.second = second

    def set\_hour(self, hour):

        self.hour = hour

    def set\_minute(self, minute):

        self.minute = minute

    def set\_seconds(self, second):

        self.second = second

    def set\_time(self, hour, minute, second):

        self.set\_hour(hour)

        self.set\_minute(minute)

        self.set\_seconds(second)

    def tick(self):

        self.\_second += 1

        if (self.\_second == 60):

            self.\_second = 0

            self.\_minute += 1

            if (self.\_minute == 60):

                self.\_minute = 0

                self.\_hour += 1

                if (self.\_hour == 24):

                    self.\_hour = 0

    def get\_time(self):

        if(self.hour > 0 and self.hour <= 12):

            state = "AM"

        elif(self.hour > 12 and self.hour < 24):

            self.hour -= 12

            state = "PM"

        elif(self.hour == 12):

            self.hour = 0

            state = "PM"

        elif(self.hour == 24):

            self.hour = 0

            state = "AM"

        print(f"{self.hour:02d}:{self.minute:02d}:{self.second:02d} {state}")

def main():

    clock = Clock(12, 50, 12)

    clock.set\_time(9,30,24)

    clock.set\_hour(13)

    clock.set\_seconds(30)

    clock.get\_time()

main()

No.2

class Poly:

    def \_\_init\_\_(self, x):

        self.x = list(x)

    def who\_is\_bigger(self, p):

        first\_power = self.find\_powerOfx()

        first\_coef = self.x

        second\_power = p.find\_powerOfx()

        second\_coef = p.x

        if (len(first\_power) >= len(second\_power)):

            bigger\_coef = first\_coef

            bigger\_power = first\_power

            smaller\_coef = second\_coef

            smaller\_power = second\_power

        elif (len(first\_power) < len(second\_power)):

            bigger\_coef = second\_coef

            bigger\_power = second\_power

            smaller\_coef = first\_coef

            smaller\_power = first\_power

        while(len(smaller\_power) != len(bigger\_power)):

            smaller\_power += (0,)

            smaller\_coef += (0,)

        return (smaller\_coef, smaller\_power, bigger\_coef, bigger\_power)

    def add(self, p):

        (smaller\_coef, smaller\_power, bigger\_coef, bigger\_power) = self.who\_is\_bigger(p)

        new\_coef = []

        for i in range(0, len(bigger\_power)):

            if bigger\_power[i] == smaller\_power[i]:

                new\_coef += (smaller\_coef[i] + bigger\_coef[i],)

            else:

                new\_coef += (bigger\_coef[i],)

        self.x = new\_coef

        return Poly(self.x)

    def scalar\_mulltiply(self, n):

        for i in range(len(self.x)):

            self.x[i] \*= n

        return Poly(self.x)

    def multiply(self, p):

        #(smaller\_coef, smaller\_power, bigger\_coef, bigger\_power) = self.who\_is\_bigger(p)

        new\_coef = []

        # print(len(self.x) + len(p.x) - 1)

        for i in range(0, (len(self.x) + len(p.x)) -1 ):

            new\_coef += (0, )

        # [0 + 0] += 1\*1

        # [0 + 1] += 1\*1

        # [1 + 0] += 1\*1

        # [1 + 1] += 1\*1

        # [2 + 0] += 1\*1

        # [2 + 1] += 1\*1

        # print(f"new\_coef = {new\_coef}")

        i = 0

        while(i != len(self.x)):

            j = 0

            while(j != len(p.x)):

                new\_coef[i+j] += (self.x[i] \* p.x[j])

                j += 1

            i += 1

        # print(f"new\_coef = {new\_coef}")

        return Poly(new\_coef)

    def power(self, n):

        new\_coef = []

        # print(len(self.x) + len(self.x))

        for i in range(0, (len(self.x) \*n ) -1 ):

            new\_coef += (0, )

        # [0 + 0] += 1\*1

        # [0 + 1] += 1\*1

        # [1 + 0] += 1\*1

        # [1 + 1] += 1\*1

        # [2 + 0] += 1\*1

        # [2 + 1] += 1\*1

        # print(f"new\_coef = {new\_coef}")

        i = 0

        while(i != len(self.x)):

            j = 0

            while(j != len(self.x)):

                new\_coef[i+j] += (self.x[i] \* self.x[j])

                j += 1

            i += 1

        # print(f"new\_coef = {new\_coef}")

        return Poly(new\_coef)

    def find\_powerOfx(self):

        count = 0

        powers = ()

        for i in self.x:

            if (i == 0):

                powers += (0,)

                count += 1

                continue

            else:

                powers += (count,)

                count += 1

        return powers

    def diff(self):

        new\_coef = ()

        for i in range(len(self.x)-1):

            new\_coef += (self.x[i + 1] \* (i + 1),)

        return Poly(new\_coef)

    def integrate(self):

        # ∫ x^n dx = (( x^(n+1) ) / (n+1))+C ; n≠1

        new\_coef = []

        # print(f"len(self.x) = {len(self.x)}")

        for i in range(0, len(self.x)+1):

            new\_coef += (0, )

        # print(f"new\_coef = {new\_coef}")

        original\_powers = self.find\_powerOfx()

        print()

        for i in range(0, len(self.x)):

            # print(f"self.x[i] / original\_powers[i] + 1  = {self.x[i] // original\_powers[i] + 1 }")

            new\_coef[i+1] = (self.x[i] / (original\_powers[i] + 1 ))

            # print(f"self.x[i] : {self.x[i]}")

            # print(f"original\_powers[i] : {original\_powers[i] + 1}")

            # print()

        # print(f"new\_coef = {new\_coef}")

        return Poly(new\_coef)

    def print(self):

        count = 0

        for i in self.x:

            sign = "+"

            if i < 0:

                sign = "-"

            if ( count == 0):

                sign = ""

            if (i == 0):

                count += 1

                continue

            else:

                i = abs(i)

                if count == 0:

                    print(f"{sign}{int(i)} ", end = "")

                else:

                    print(f"{sign} {int(i)}x^{count} ", end = " ")

                count += 1

        print()

    def eval(self, n):

        count = 0

        total = 0

        for i in self.x:

            if (i == 0):

                # print(f"Skipped {count}^th power since it's coeff is 0.")

                count += 1

                continue

            else:

                # print(f"total = {total}")

                total += i \* (n \*\* count)

                # print(f"{i} \* ({n} \*\* {count}) = {total}")

                count += 1

        print(total)

def main():

    #          0th, 1th, 2th, 3th, 4th, ....

    p = Poly( (  1, 2, 3) )

    p.print()

    print()

    p.diff().print()

    p.integrate().print()

    # q = p.power(2)

    # q.print()

    # p.print()

    # p.eval(3)

    # print(p.find\_powerOfx())

    # print(q.find\_powerOfx())

    # r = p.add(q)

    # r.print()

    # p.scalar\_mulltiply(2)

    # p.print()

    # s = Poly((1,1))

    # s.print()

    # r = p.multiply(s)

    # r.print()

    # p.diff()

    # print(f"Diff of p : ", end = "")

    # p.print()

    # r = p.power(2)

    # r.print()

main()

No.3

class LinearEquation:

    def \_\_init\_\_(self, a, b, c, d, e, f):

        self.\_\_a = a

        self.\_\_b = b

        self.\_\_c = c

        self.\_\_d = d

        self.\_\_e = e

        self.\_\_f = f

    def get\_a(self, a):

        return self.\_\_a

    def get\_b(self, b):

        return self.\_\_b

    def get\_c(self, c):

        return self.\_\_c

    def get\_d(self, d):

        return self.\_\_d

    def get\_e(self, e):

        return self.\_\_e

    def get\_f(self, f):

        return self.\_\_f

    def isSolvable(self):

        if( (self.\_\_a \* self.\_\_d) - (self.\_\_b\* self.\_\_c) !=0 ):

            return True

        else:

            return False

    def getX(self):

        top =( self.\_\_e \* self.\_\_d) - (self.\_\_b \* self.\_\_f)

        bottom = (self.\_\_a \* self.\_\_d) - (self.\_\_b \* self.\_\_c)

        return top / bottom

    def getY(self):

        top =( self.\_\_a \* self.\_\_f) - (self.\_\_e \* self.\_\_c)

        bottom = (self.\_\_ \* self.\_\_d) - (self.\_\_b \* self.\_\_c)

        return top / bottom