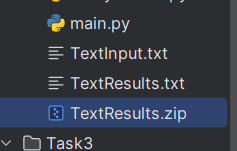
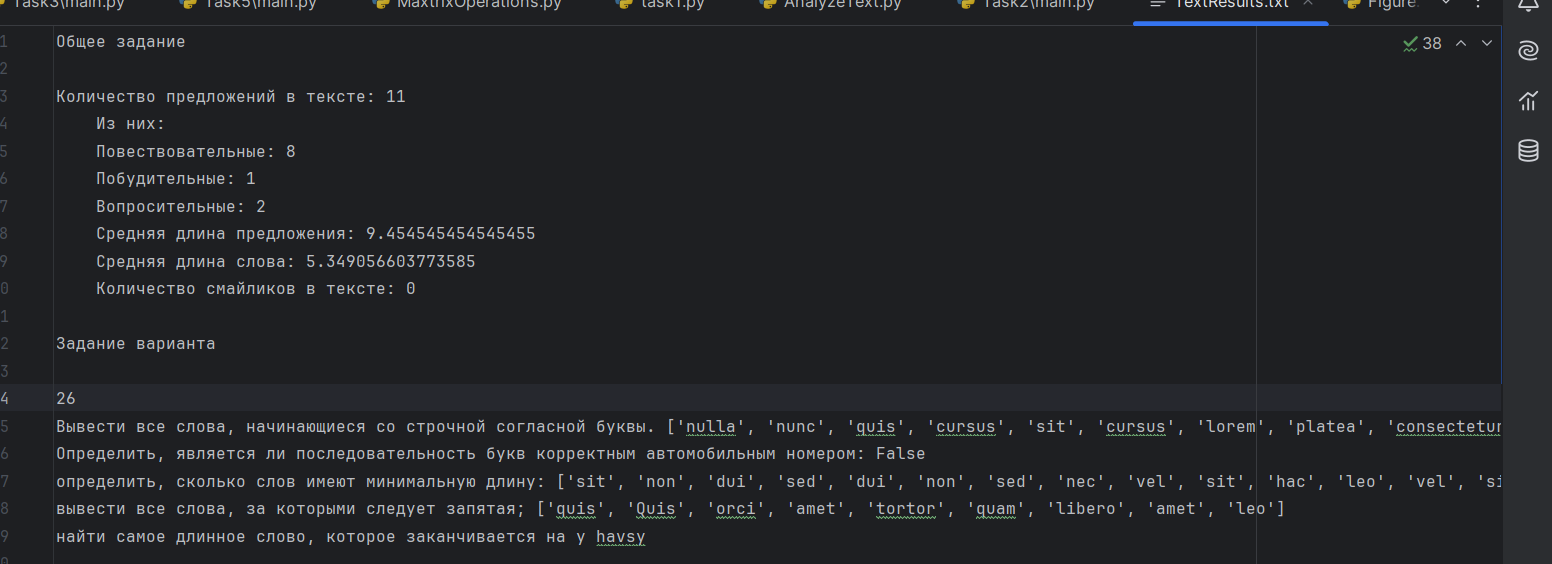
import csv  
import pickle  
  
class Forest:  
 def \_\_init\_\_(self):  
 self.trees = {}  
  
 def add\_tree(self, species, total\_count, healthy\_count):  
 if species in self.trees:  
 print("Дерево с таким видом уже существует в лесу.")  
 return  
 self.trees[species] = {'total\_count': total\_count, 'healthy\_count': healthy\_count}  
  
 def total\_trees(self):  
 return sum(tree['total\_count'] for tree in self.trees.values())  
  
 def total\_healthy\_trees(self):  
 return sum(tree['healthy\_count'] for tree in self.trees.values())  
  
 def relative\_sickness\_percentage(self):  
 total\_sick\_trees = self.total\_trees() - self.total\_healthy\_trees()  
 return (total\_sick\_trees / self.total\_trees()) \* 100 if self.total\_trees() > 0 else 0  
  
 def species\_percentage(self, species):  
 if species not in self.trees:  
 print("Дерево с таким видом не найдено.")  
 return 0, 0  
 total\_species\_count = self.trees[species]['total\_count']  
 total\_species\_healthy\_count = self.trees[species]['healthy\_count']  
 species\_percentage = (total\_species\_count / self.total\_trees()) \* 100  
 species\_sickness\_percentage = ((total\_species\_count - total\_species\_healthy\_count) / total\_species\_count) \* 100  
 return species\_percentage, species\_sickness\_percentage  
  
 def find\_tree(self, species):  
 if species in self.trees:  
 return self.trees[species]  
 else:  
 return None  
  
 def serialize\_to\_csv(self, filename):  
 with open(filename, mode='w', newline='') as file:  
 writer = csv.writer(file)  
 writer.writerow(['Species', 'Total Count', 'Healthy Count'])  
 for species, data in self.trees.items():  
 writer.writerow([species, data['total\_count'], data['healthy\_count']])  
  
 @classmethod  
 def deserialize\_from\_csv(cls, filename):  
 forest = cls()  
 with open(filename, mode='r') as file:  
 reader = csv.DictReader(file)  
 for row in reader:  
 species = row['Species']  
 total\_count = int(row['Total Count'])  
 healthy\_count = int(row['Healthy Count'])  
 forest.add\_tree(species, total\_count, healthy\_count)  
 return forest  
  
 def serialize\_to\_pickle(self, filename):  
 with open(filename, 'wb') as file:  
 pickle.dump(self.trees, file)  
  
 @classmethod  
 def deserialize\_from\_pickle(cls, filename):  
 forest = cls()  
 with open(filename, 'rb') as file:  
 forest.trees = pickle.load(file)  
 return forest  
  
def main():  
 initial\_data = {  
 "Oak": {"total\_count": 100, "healthy\_count": 80},  
 "Pine": {"total\_count": 150, "healthy\_count": 100},  
 "Maple": {"total\_count": 80, "healthy\_count": 60}  
 }  
  
 forest = Forest()  
 for species, data in initial\_data.items():  
 forest.add\_tree(species, data["total\_count"], data["healthy\_count"])  
  
 print("Суммарное число деревьев на контрольном участке:", forest.total\_trees())  
 print("Суммарное число здоровых деревьев:", forest.total\_healthy\_trees())  
 print("Относительная численность (%) больных деревьев:", forest.relative\_sickness\_percentage())  
  
 for species in initial\_data.keys():  
 species\_percentage, sickness\_percentage = forest.species\_percentage(species)  
 print(f"Относительная численность (%) виду {species}: {species\_percentage:.2f}%")  
 print(f"Относительная численность (%) больных деревьев виду {species}: {sickness\_percentage:.2f}%")  
  
 search\_species = input("Введите вид дерева для поиска: ")  
 found\_tree = forest.find\_tree(search\_species)  
 if found\_tree:  
 print(f"Информация о дереве {search\_species}:")  
 print("Общее число деревьев:", found\_tree['total\_count'])  
 print("Число здоровых деревьев:", found\_tree['healthy\_count'])  
 else:  
 print(f"Дерево с видом {search\_species} не найдено.")  
  
  
  
 # Сериализация и запись в файлы CSV и pickle  
 forest.serialize\_to\_csv('forest.csv')  
 forest.serialize\_to\_pickle('forest.pickle')  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

import re  
import zipfile  
  
class TextAnalyzer:  
 def \_\_init\_\_(self, text=''):  
 self.\_\_text = text  
  
 @property  
 def text(self):  
 return self.\_\_text  
  
 @text.setter  
 def text(self, input\_text):  
 self.\_\_text = input\_text  
  
 def sentences\_count(self):  
 *"""Count the number of sentences in the text."""* sentences = re.findall(r'[A-Z0-9][^?!.]\*[!?.]', self.\_\_text)  
 return len(sentences)  
  
 def declarative\_sentences\_count(self):  
 *"""Count the number of declarative sentences in the text."""* sentences = re.findall(r'[A-Z0-9][^?!.]\*[.]', self.\_\_text)  
 return len(sentences)  
  
 def exclamatory\_sentences\_count(self):  
 *"""Count the number of exclamatory sentences in the text."""* sentences = re.findall(r'[A-Z0-9][^?!.]\*[!]', self.\_\_text)  
 return len(sentences)  
  
 def interrogative\_sentences\_count(self):  
 *"""Count the number of interrogative sentences in the text."""* sentences = re.findall(r'[A-Z0-9][^?!.]\*[?]', self.\_\_text)  
 return len(sentences)  
  
 def average\_sentence\_length(self):  
 *"""Calculate the average length of sentences (in words) in the text."""* sentences = re.findall(r'[A-Z0-9][^?!.]\*[!?.]', self.\_\_text)  
 word\_count = sum(len(snt.split()) for snt in sentences)  
 return word\_count / len(sentences) if len(sentences) > 0 else 0  
  
 def average\_word\_length(self):  
 *"""Calculate the average length of words in the text (excluding punctuation)."""* words = re.findall(r'\b\w+\b', self.\_\_text)  
 word\_count = len(words)  
 total\_length = sum(len(word) for word in words)  
 return total\_length / word\_count if word\_count > 0 else 0  
  
 def emojis\_count(self):  
 *"""Count the number of emojis in the text."""* emojis = re.findall(r'[:;]-\*([(\[]+)', self.\_\_text)  
 return len(emojis)  
  
 def lowercase\_consonant\_words(self):  
 # Вывод всех слов, начинающихся со строчной согласной буквы  
 words = re.findall(r'\b[b-df-hj-np-tv-z]\w\*\b', self.\_\_text)  
 return words  
  
 def is\_car\_number(self):  
 # Определение, является ли последовательность букв корректным автомобильным номером  
 car\_number\_pattern = r'[A-Z]{2}\d{4}[A-Z]{2}\d'  
 return bool(re.match(car\_number\_pattern, self.\_\_text))  
  
 def words\_with\_min\_length(self, min\_length):  
 # Подсчет количества слов с минимальной длиной  
 words = re.findall(r'\b\w+\b', self.\_\_text)  
 min\_length\_words = [word for word in words if len(word) == min\_length]  
 return min\_length\_words  
  
 def words\_followed\_by\_comma(self):  
 # Вывод всех слов, за которыми следует запятая  
 words = re.findall(r'\b\w+\b(?=,)', self.\_\_text)  
 return words  
  
 def longest\_word\_ending\_with\_y(self):  
 # Поиск самого длинного слова, которое заканчивается на 'y'  
 words = re.findall(r'\b\w+[y]\b', self.\_\_text)  
 if words:  
 return max(words, key=len)  
 else:  
 return None  
  
  
  
 def execute\_main\_analysis(self):  
 *"""Write main task results to a file."""* with open('TextResults.txt', 'w', encoding='utf-8') as file:  
 file.write('Общее задание\n\n')  
 file.write(f'Количество предложений в тексте: {self.sentences\_count()}\n')  
 file.write('\tИз них:\n')  
 file.write(f'\tПовествовательные: {self.declarative\_sentences\_count()}\n')  
 file.write(f'\tПобудительные: {self.exclamatory\_sentences\_count()}\n')  
 file.write(f'\tВопросительные: {self.interrogative\_sentences\_count()}\n')  
 file.write(f'\tСредняя длина предложения: {self.average\_sentence\_length()}\n')  
 file.write(f'\tСредняя длина слова: {self.average\_word\_length()}\n')  
 file.write(f'\tКоличество смайликов в тексте: {self.emojis\_count()}\n')  
 file.close()  
  
 def execute\_additional\_analysis(self):  
 *"""Write additional task results to a file """* with open('TextResults.txt', 'a', encoding='utf-8') as file:  
 file.write('\nЗадание варианта\n\n')  
 file.write('26\n')  
 file.write(f'Вывести все слова, начинающиеся со строчной согласной буквы. {self.lowercase\_consonant\_words()}\n')  
 file.write(f'Определить, является ли последовательность букв корректным автомобильным номером: {self.is\_car\_number()}\n')  
 file.write(f'определить, сколько слов имеют минимальную длину: {self.words\_with\_min\_length(3)}\n')  
 file.write(f'вывести все слова, за которыми следует запятая; {self.words\_followed\_by\_comma()}\n')  
 file.write(f'найти самое длинное слово, которое заканчивается на y {self.longest\_word\_ending\_with\_y()}\n')  
 file.close()  
  
 with zipfile.ZipFile('TextResults.zip', 'w') as zip\_file:  
 zip\_file.write('TextResults.txt', arcname='TextResults.txt')  
  
 print(f'Содержимое архива TextResults.zip:')  
 for info in zip\_file.infolist():  
 print(f'Имя файла: {info.filename}')  
 print(f'Размер файла: {info.file\_size} байт')  
 print(f'Сжатый размер: {info.compress\_size} байт')  
 print(f'Дата модификации: {info.date\_time}')  
 print('---')  
 zip\_file.close()

from Task2.AnalyzeText import TextAnalyzer  
  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 text\_op = TextAnalyzer()  
 with open('TextInput.txt', 'r', encoding='utf-8') as file:  
 text = file.read()  
 text\_op.text = text  
  
 text\_op.execute\_main\_analysis()  
 text\_op.execute\_additional\_analysis()  
  
 print("Задачи выполнены. Результаты сохранены в файлы TextResults.txt и TextResults.zip.")

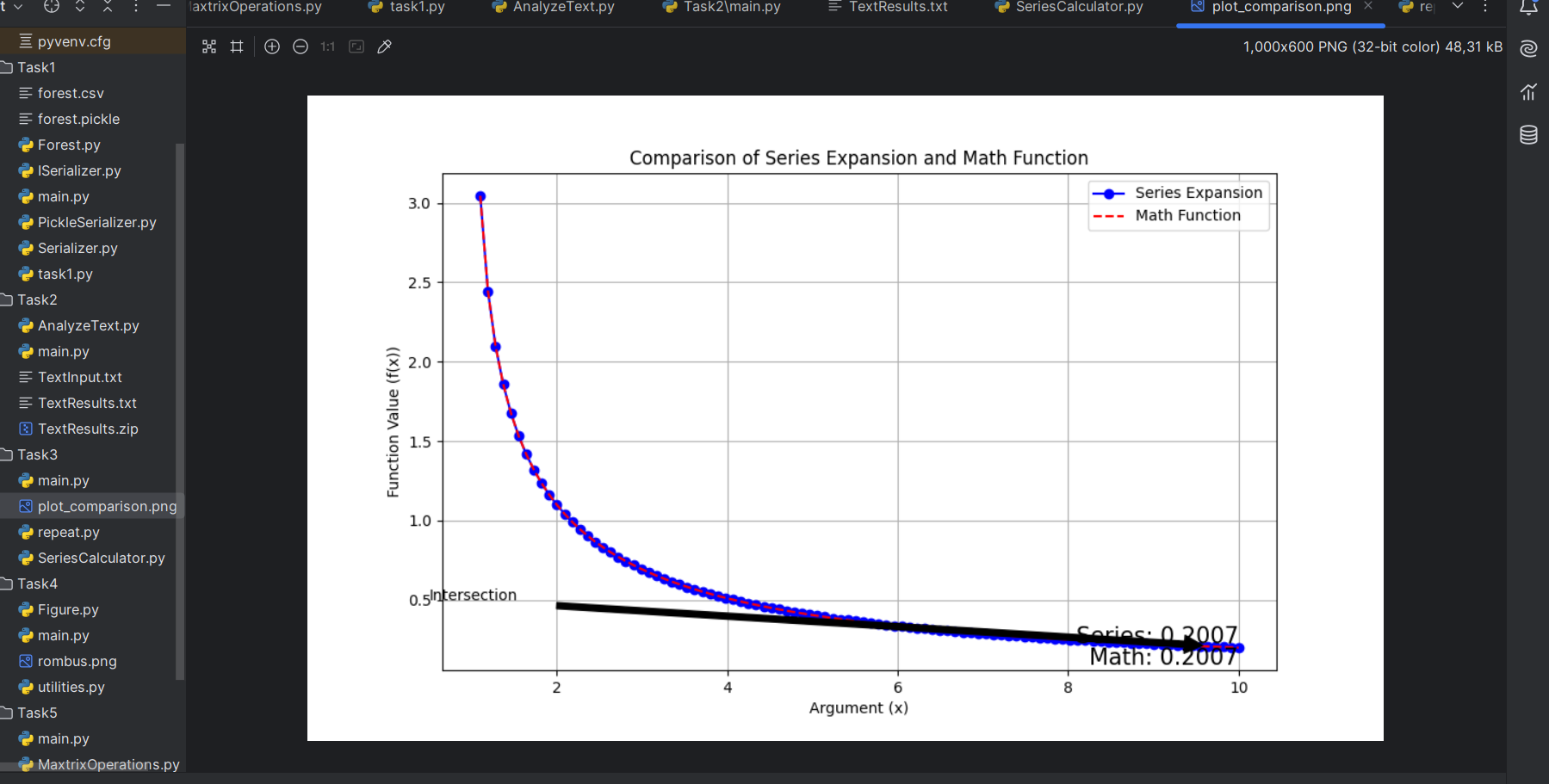




import math  
import numpy as np  
from SeriesCalculator import SeriesCalculator  
import matplotlib.pyplot as plt  
from repeat import should\_repeat  
  
  
def plot\_graphs(x\_values, series\_values, math\_function\_values, save\_path=None):  
 plt.figure(figsize=(10, 6))  
 plt.plot(x\_values, series\_values, label='Series Expansion', color='blue', marker='o')  
 plt.plot(x\_values, math\_function\_values, label='Math Function', color='red', linestyle='--')  
 plt.xlabel('Argument (x)')  
 plt.ylabel('Function Value (f(x))')  
 plt.title('Comparison of Series Expansion and Math Function')  
 plt.legend()  
 plt.grid(True)  
  
 # Annotate with text  
 plt.text(x\_values[-1], series\_values[-1], f'Series: {series\_values[-1]:.4f}', fontsize=15, ha='right', va='bottom')  
 plt.text(x\_values[-1], math\_function\_values[-1], f'Math: {math\_function\_values[-1]:.4f}', fontsize=15, ha='right',  
 va='top')  
  
 # Add annotation  
 plt.annotate('Intersection', xy=(x\_values[-1], series\_values[-1]), xytext=(0.5, 0.5),  
 arrowprops=dict(facecolor='black', shrink=0.05))  
  
 if save\_path:  
 plt.savefig(save\_path)  
 print(f"Plot saved successfully at: {save\_path}")  
 else:  
 plt.show()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 calculator = SeriesCalculator()  
 while True:  
 x = float(input("Enter the value of the argument (x), where |x| > 1: "))  
 eps = float(input("Enter the desired accuracy (eps): "))  
   
 if abs(x) <= 1:  
 print("The absolute value of x must be greater than 1.")  
 continue  
 if eps <= 0:  
 print("Accuracy (eps) must be a positive number.")  
 continue  
  
 result, n, terms, mean, median, mode, variance, std\_dev = calculator.ln\_series(x, eps)  
  
 math\_result = math.log((x + 1) / (x - 1))  
  
 print("\nResults:")  
 print(f"| x = {x} | n = {n} | F(x) = {result} | mathF = {math\_result} | eps = {eps}")  
 print(f"Mean: {mean:.4f}")  
 print(f"Median: {median:.4f}")  
 print(f"Mode: {mode:.4f}")  
 print(f"Variance: {variance:.4f}")  
 print(f"Standard Deviation: {std\_dev:.4f}")  
  
 # Generate x values for plotting  
 x\_values = np.linspace(1.1, x, 100)  
 series\_values = [calculator.ln\_series(val, eps)[0] for val in x\_values]  
 math\_function\_values = [math.log((val + 1) / (val - 1)) for val in x\_values]  
  
 save\_path = "plot\_comparison.png"  
 # Plotting graphs  
 plot\_graphs(x\_values, series\_values, math\_function\_values, save\_path)  
  
 if not should\_repeat():  
 break

def should\_repeat():  
 repeat = input("\nDo you want to perform the task again? (yes/no): ")  
 return repeat.lower() == 'yes'

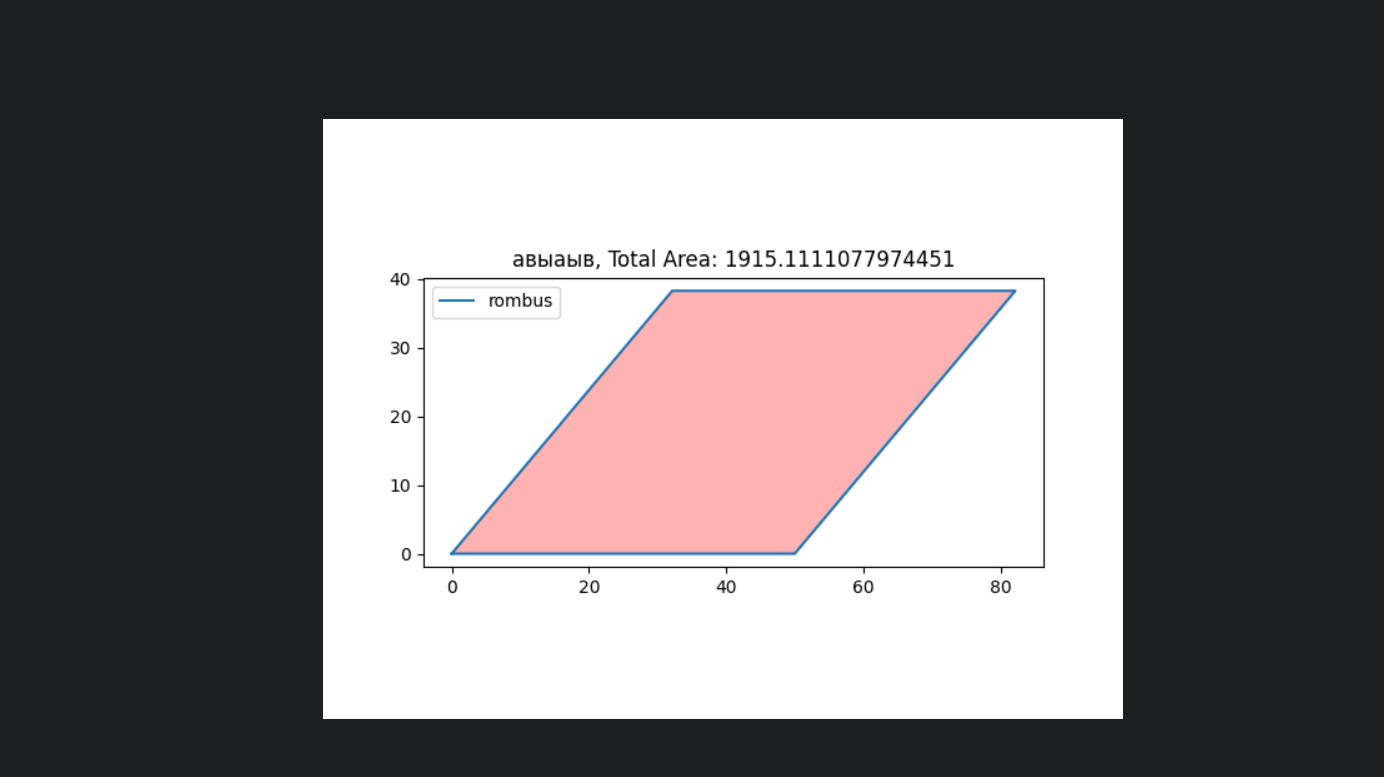
import math  
import numpy as np  
  
  
  
default\_max\_iterations = 500  
  
class SeriesCalculator():  
 def \_\_init\_\_(self):  
 pass  
  
 def ln\_series(self, x, eps, max\_iterations=default\_max\_iterations):  
 *"""  
 Function to compute the value of ln((x+1)/(x-1)) using power series expansion.  
 """* result = 0  
 terms = []  
 for n in range(max\_iterations):  
 term = 1 / ((2 \* n + 1) \* math.pow(x, 2 \* n + 1))  
 result += term  
 terms.append(result \* 2)  
 if abs(term) < eps:  
 break  
   
 # Compute statistical parameters  
 sequence = np.array(terms)  
 mean = np.mean(sequence)  
 median = np.median(sequence)  
 if len(sequence) > 0:  
 unique\_values, counts = np.unique(sequence, return\_counts=True)  
 mode\_index = np.argmax(counts)  
 mode\_value = unique\_values[mode\_index]  
 else:  
 mode\_value = None  
  
 variance = np.var(sequence)  
 std\_dev = np.std(sequence)  
  
 return result \* 2, n, terms, mean, median, mode\_value, variance, std\_dev



import math  
import matplotlib.pyplot as plt  
from abc import ABC, abstractmethod  
  
class NameMixin:  
 def \_\_init\_\_(self, name):  
 self.\_name = name  
  
 @property  
 def name(self):  
 return self.\_name  
  
 @name.setter  
 def name(self, new\_name):  
 self.\_name = new\_name  
  
class Color:  
 def \_\_init\_\_(self, color):  
 *"""Initialize color."""* self.\_color = color  
  
 @property  
 def color(self):  
 *"""Getter for color."""* return self.\_color  
  
 @color.setter  
 def color(self, new\_color):  
 *"""Setter for color."""* self.\_color = new\_color  
  
 def \_\_str\_\_(self):  
 *"""String representation of Color object."""* return self.\_color  
  
class GeometricFigure(ABC):  
 @abstractmethod  
 def area(self):  
 *"""Abstract method for calculating area."""* pass  
  
class Romb(GeometricFigure, NameMixin):  
 def \_\_init\_\_(self, name, length, angle\_degrees, color):  
 NameMixin.\_\_init\_\_(self, name) # Инициализация имени через миксин  
 self.color = Color(color)  
 self.length = length  
 self.angle\_radians = math.radians(angle\_degrees)  
  
 def area(self):  
 *"""Calculate the area of the rombus."""* return ((self.length \* self.length) \* math.sin(self.angle\_radians))  
  
 def \_\_str\_\_(self):  
 *"""Formatted string representation of the figure."""* return "Name: {}\nColor: {}\nArea: {:.2f}".format(self.name, self.color, self.area())  
  
 def draw(self):  
 *"""Draw the isosceles rombus."""* fig, ax = plt.subplots()  
  
 # Calculate the coordinates of the vertices  
 x1 = 0  
 y1 = 0  
 x2 = self.length  
 y2 = 0  
 x3 = self.length + self.length \*math.cos(self.angle\_radians)  
 y3 = self.length \* math.sin(self.angle\_radians)  
 x4 = self.length \*math.cos(self.angle\_radians)  
 y4 = self.length \* math.sin(self.angle\_radians)  
  
 # Plot the rombus  
 rombus\_coords = [(x1, y1), (x2, y2), (x3, y3), (x4, y4), (x1, y1)]  
 ax.plot(\*zip(\*rombus\_coords), label='rombus')  
  
 # Fill the rombus  
 plt.fill(\*zip(\*rombus\_coords), color=self.color.color, alpha=0.3)  
  
 ax.set\_aspect('equal', adjustable='box')  
 ax.set\_title(f"{self.name}, Total Area: {self.area()}")  
 plt.legend()  
  
 # Save the image to rombus.png in the current directory  
 plt.savefig('rombus.png')  
  
 # Display the image  
 plt.show()

from Figure import Romb  
from utilities import Repeat  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 while True:  
 try:  
 name = input("Enter the name of the figure: ")  
 base\_length = float(input("Enter the length of the base of the rombus: "))  
 angle\_degrees = float(input("Enter the angle between the base and the side in degrees: "))  
 color = input("Enter the color of the rombus ('red', 'blue', '#FFA500', etc.): ")  
  
 if base\_length <= 0:  
 raise ValueError("Lengths must be positive numbers.")  
  
 if angle\_degrees <= 0 or angle\_degrees >= 180:  
 raise ValueError("Angle must be between 0 and 180 degrees.")  
  
 figure = Romb(name, base\_length, angle\_degrees, color)  
 print(figure)  
 figure.draw()  
  
 if not Repeat():  
 break  
  
 except ValueError as e:  
 print(f"Error: {e}. Please enter valid data.")

def Repeat():  
 repeat = input("\nDo you want to perform the task again? (yes/no): ")  
 return repeat.lower() == 'yes'



import numpy as np  
from MaxtrixOperations import MatrixOperations  
from repeat import should\_repeat  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 while True:   
 n = 4  
 m = 5  
 B = 50  
 matrix\_ops = MatrixOperations(n, m)  
 matrix\_ops.generate\_matrix()  
  
 print("Generated Matrix:")  
 print(matrix\_ops.matrix)  
 print()  
  
 C, count\_elements = matrix\_ops.find\_elements\_above\_abs(B)  
 print("Number of elements exceeding absolute value", B, ":", count\_elements)  
 print("Elements exceeding absolute value", B, ":")  
 print(C)  
 print()  
  
 median\_C = matrix\_ops.calculate\_median(C)  
 print("Median of array C (using custom method):", median\_C)  
 print("Median of array C (using np.median):", np.median(C))  
 print()  
  
 print("Mean of matrix A:", matrix\_ops.mean())  
 print("Median of matrix A:", matrix\_ops.median())  
 print("Correlation coefficient matrix of matrix A:")  
 print(matrix\_ops.corrcoef())  
 print("Variance of matrix A:", matrix\_ops.variance())  
 print("Standard deviation of matrix A:", matrix\_ops.std\_deviation())  
  
 if not should\_repeat():  
 break

import numpy as np  
  
  
class MatrixOperations:  
 def \_\_init\_\_(self, n, m):  
 *"""  
 Initialize MatrixOperations with matrix dimensions n x m.  
 """* self.n = n  
 self.m = m  
 self.matrix = None  
  
 def generate\_matrix(self):  
 *"""  
 Generate a random matrix of size n x m with integer values.  
 """* self.matrix = np.random.randint(0, 100, size=(self.n, self.m))  
  
 def find\_elements\_above\_abs(self, B):  
 *"""  
 Find elements in the matrix exceeding the absolute value of B.  
 Return these elements in an array C and count the number of such elements.  
 """* if self.matrix is None:  
 print("Matrix not generated. Please call generate\_matrix() first.")  
 return None, 0  
   
 mask = np.abs(self.matrix) > B  
 C = self.matrix[mask]  
 count\_elements = len(C)  
 return C, count\_elements  
  
 def calculate\_median(self, array):  
 *"""  
 Calculate the median of the given array.  
 """* if array is None or len(array) == 0:  
 return None  
   
 sorted\_array = np.sort(array)  
 n = len(sorted\_array)  
 if n % 2 == 1:  
 median = sorted\_array[n // 2]  
 else:  
 median = (sorted\_array[n // 2 - 1] + sorted\_array[n // 2]) / 2  
 return median  
  
 def mean(self):  
 *"""  
 Compute the mean of all elements in the matrix.  
 """* if self.matrix is None:  
 print("Matrix not generated. Please call generate\_matrix() first.")  
 return None  
   
 return np.mean(self.matrix)  
  
 def median(self):  
 *"""  
 Compute the median of all elements in the matrix.  
 """* if self.matrix is None:  
 print("Matrix not generated. Please call generate\_matrix() first.")  
 return None  
   
 return np.median(self.matrix)  
  
 def corrcoef(self):  
 *"""  
 Compute the correlation coefficient matrix of the matrix.  
 """* if self.matrix is None:  
 print("Matrix not generated. Please call generate\_matrix() first.")  
 return None  
   
 return np.corrcoef(self.matrix)  
  
 def variance(self):  
 *"""  
 Compute the variance (dispersion) of all elements in the matrix.  
 """* if self.matrix is None:  
 print("Matrix not generated. Please call generate\_matrix() first.")  
 return None  
   
 return np.var(self.matrix)  
  
 def std\_deviation(self):  
 *"""  
 Compute the standard deviation of all elements in the matrix.  
 """* if self.matrix is None:  
 print("Matrix not generated. Please call generate\_matrix() first.")  
 return None  
   
 return np.std(self.matrix)

def should\_repeat():  
 repeat = input("\nDo you want to perform the task again? (yes/no): ")  
 return repeat.lower() == 'yes'

