Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра Информатики

Дисциплина «Избранные главы информатики»

**ОТЧЕТ**

к лабораторной работе №4

на тему:

**«Работа с файлами, классами, сериализаторами, регулярными выражениями и стандартными библиотеками.»**

БГУИР 6-05-0612-02 01

|  |
| --- |
| Выполнила студентка группы 353504  АНТОНОВА Лидия Сергеевна |
|  |
| (дата, подпись студента) |
| Проверила  ЖВАКИНА Анна Васильевна |
|  |
| (дата, подпись преподавателя) |

Минск 2025

# Цель работы

Освоить базовый синтаксис языка Python, приобрести навыки работы с файлами, классами, сериализаторами, регулярными выражениями и стандартными библиотеками и закрепить их на примере разработки интерактивных приложений.

Вариант 1.

# Выполнение работы

Задание 1. Исходные данные представляют собой словарь. Необходимо поместить их в файл, используя сериализатор. Организовать считывание данных, поиск, сортировку в соответствии с индивидуальным заданием. Обязательно использовать классы. Реализуйте два варианта: 1) формат файлов CSV; 2) модуль pickle.

В сводке об экспортируемых товарах указывается: наименование товара, страна, импортирующая товар, объем поставляемой партии в штуках. Напечатайте списки стран, в которые экспортируется данный товар, и общий объем его экспорта. Выведите информацию о товаре, введенном с клавиатуры.

from .serializer import LoggingMixin  
  
class BaseExportProduct:  
 *"""  
 Base class representing export product data.  
 Contains static attribute 'count' to track the number of product instances.  
 """* count = 0  
  
 def \_\_init\_\_(self, name):  
 *"""  
 Initialize with product name.  
 Args:  
 name (str): The name of the product.  
 """* self.\_name = name  
 self.import\_records = []  
 BaseExportProduct.count += 1  
  
 @property  
 def name(self):  
 *"""Getter for the product name."""* return self.\_name  
  
 @name.setter  
 def name(self, value):  
 *"""  
 Setter for the product name.  
 Args:  
 value (str): The new product name.  
 Raises:  
 ValueError: If the provided name is not a non-empty string.  
 """* if isinstance(value, str) and value:  
 self.\_name = value  
 else:  
 raise ValueError("Name must be a non-empty string")  
  
 def add\_record(self, country, volume):  
 *"""  
 Add an import record to the product.  
 Args:  
 country (str): Country importing the product.  
 volume (int): Volume of the product shipment.  
 """* self.import\_records.append((country, volume))  
  
 def total\_volume(self):  
 *"""  
 Calculate the total shipment volume of the product.  
 Returns:  
 int: Total volume.  
 """* return sum(volume for \_, volume in self.import\_records)  
  
 def importing\_countries(self):  
 *"""  
 Get a list of unique importing countries for the product.  
 Returns:  
 list: List of country names.  
 """* return list(set(country for country, \_ in self.import\_records))  
  
 def \_\_str\_\_(self):  
 *"""  
 Magic method to return a formatted string representing product details.  
 Returns:  
 str: Formatted product information.  
 """* return f"Product: {self.name}, Total Volume: {self.total\_volume()}, Countries: {self.importing\_countries()}"  
  
 def \_\_eq\_\_(self, other):  
 *"""  
 Check equality based on product name.  
 Args:  
 other (BaseExportProduct): Another product instance.  
 Returns:  
 bool: True if product names are equal, False otherwise.  
 """* if isinstance(other, BaseExportProduct):  
 return self.name == other.name  
 return False  
  
class DetailedExportProduct(BaseExportProduct, LoggingMixin):  
 *"""  
 Derived class that adds detailed logging functionality.  
 Demonstrates usage of super() and mixins.  
 """* def \_\_init\_\_(self, name, category="General"):  
 *"""  
 Initialize with product name and category.  
 Args:  
 name (str): The name of the product.  
 category (str): Category of the product.  
 """* super().\_\_init\_\_(name)  
 self.category = category # dynamic attribute in the subclass  
  
 def add\_record(self, country, volume):  
 *"""  
 Override add\_record to include logging functionality.  
 Args:  
 country (str): Country importing the product.  
 volume (int): Volume of the product shipment.  
 """* self.log(f"Adding record for {self.name}: {country} - {volume}")  
 super().add\_record(country, volume)  
  
 def \_\_str\_\_(self):  
 *"""  
 Override \_\_str\_\_ method to include category information.  
 Returns:  
 str: Formatted detailed product information.  
 """* base\_str = super().\_\_str\_\_()  
 return f"{base\_str}, Category: {self.category}"

import csv  
import pickle  
  
class LoggingMixin:  
 *"""  
 A Mixin class providing logging functionality.  
 """* def log(self, message):  
 *"""Simple logging method."""* print(f"[LOG]: {message}")  
  
class BaseSerializer:  
 *"""  
 Base serializer class defining the interface for serialization.  
 """* def save(self, data, filename):  
 *"""Save data to a file."""* raise NotImplementedError("Save method not implemented")  
  
 def load(self, filename):  
 *"""Load data from a file."""* raise NotImplementedError("Load method not implemented")  
  
class CSVSerializer(BaseSerializer, LoggingMixin):  
 *"""  
 Serializer for CSV format.  
 Demonstrates usage of static and dynamic attributes.  
 """* delimiter = ','  
  
 def save(self, data, filename):  
 *"""  
 Save dictionary data to a CSV file.  
 Args:  
 data (dict): Data to be saved (expected format: {product: list of records}).  
 filename (str): Name of the CSV file.  
 """* try:  
 with open(filename, mode='w', newline='', encoding='utf-8') as csvfile:  
 writer = csv.writer(csvfile, delimiter=self.delimiter)  
  
 writer.writerow(["Product", "Country", "Volume"])  
 for product, records in data.items():  
 for record in records:  
 writer.writerow([product, record["country"], record["volume"]])  
 self.log(f"Data successfully saved to CSV file: {filename}")  
 except Exception as e:  
 self.log(f"Error saving CSV: {e}")  
 raise  
  
 def load(self, filename):  
 *"""  
 Load data from a CSV file and return in dictionary format.  
 Returns:  
 dict: Data in format {product: list of records}.  
 """* result = {}  
 try:  
 with open(filename, mode='r', newline='', encoding='utf-8') as csvfile:  
 reader = csv.DictReader(csvfile, delimiter=self.delimiter)  
 for row in reader:  
 product = row["Product"]  
 country = row["Country"]  
 volume = int(row["Volume"])  
 if product in result:  
 result[product].append({"country": country, "volume": volume})  
 else:  
 result[product] = [{"country": country, "volume": volume}]  
 self.log(f"Data successfully loaded from CSV file: {filename}")  
 return result  
 except Exception as e:  
 self.log(f"Error loading CSV: {e}")  
 raise  
  
class PickleSerializer(BaseSerializer, LoggingMixin):  
 *"""  
 Serializer using the Python pickle module.  
 """* def save(self, data, filename):  
 *"""  
 Save data using pickle serialization.  
 Args:  
 data (dict): Data to be saved.  
 filename (str): Name of the pickle file.  
 """* try:  
 with open(filename, "wb") as pfile:  
 pickle.dump(data, pfile)  
 self.log(f"Data successfully saved to pickle file: {filename}")  
 except Exception as e:  
 self.log(f"Error saving pickle: {e}")  
 raise  
  
 def load(self, filename):  
 *"""  
 Load data using pickle deserialization.  
 Returns:  
 dict: The loaded data.  
 """* try:  
 with open(filename, "rb") as pfile:  
 data = pickle.load(pfile)  
 self.log(f"Data successfully loaded from pickle file: {filename}")  
 return data  
 except Exception as e:  
 self.log(f"Error loading pickle: {e}")  
 raise

from .export\_product import DetailedExportProduct  
  
def search\_product(data, product\_name):  
 *"""  
 Search for a product in the data dictionary.  
 Args:  
 data (dict): Dictionary containing export data.  
 product\_name (str): The product name to search for.  
 Returns:  
 list or None: List of records if found, otherwise None.  
 """* return data.get(product\_name, None)  
  
def sort\_products\_by\_total\_volume(data):  
 *"""  
 Sort products based on total volume in descending order.  
 Args:  
 data (dict): Dictionary with product names as keys and list of record dicts as values.  
 Returns:  
 list: List of tuples (product\_name, total\_volume) sorted by volume descending.  
 """* products = []  
 for product, records in data.items():  
 total = sum(record["volume"] for record in records)  
 products.append((product, total))  
 products.sort(key=lambda x: x[1], reverse=True)  
 return products  
  
def create\_product\_objects(data):  
 *"""  
 Create DetailedExportProduct objects from the data dictionary.  
 Args:  
 data (dict): Dictionary with product names as keys and list of record dicts as values.  
 Returns:  
 dict: Mapping of product names to DetailedExportProduct instances.  
 """* products = {}  
 for product\_name, records in data.items():  
 prod\_obj = DetailedExportProduct(product\_name)  
 for record in records:  
 prod\_obj.add\_record(record["country"], record["volume"])  
 products[product\_name] = prod\_obj  
 return products

import os  
from .serializer import CSVSerializer, PickleSerializer  
from .utils import search\_product, sort\_products\_by\_total\_volume, create\_product\_objects  
  
def task1():  
 *"""  
 Function representing Task 1.  
 Offers functionality to save/load export data to CSV and Pickle,  
 perform search, sorting, and display product export information.  
 """* export\_data = {  
 "Apple": [{"country": "Germany", "volume": 1000}, {"country": "France", "volume": 500}],  
 "Orange": [{"country": "Italy", "volume": 1500}, {"country": "Spain", "volume": 700}],  
 "Banana": [{"country": "Belgium", "volume": 800}, {"country": "Netherlands", "volume": 600}]  
 }  
  
 current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
  
 data\_dir = os.path.join(current\_dir, "data")  
 os.makedirs(data\_dir, exist\_ok=True)  
  
 csv\_filename = os.path.join(data\_dir, "export\_data.csv")  
 pickle\_filename = os.path.join(data\_dir, "export\_data.pkl")  
  
 csv\_serializer = CSVSerializer()  
 pickle\_serializer = PickleSerializer()  
  
 try:  
 csv\_serializer.save(export\_data, csv\_filename)  
 pickle\_serializer.save(export\_data, pickle\_filename)  
 except Exception as e:  
 print(f"Error during serialization: {e}")  
 return  
  
 print("Select file format to load data:")  
 print("1. CSV")  
 print("2. Pickle")  
  
 choice = input("Enter choice (1 or 2): ").strip()  
 if choice == "1":  
 try:  
 loaded\_data = csv\_serializer.load(csv\_filename)  
 except Exception as e:  
 print(f"Error loading CSV data: {e}")  
 return  
 elif choice == "2":  
 try:  
 loaded\_data = pickle\_serializer.load(pickle\_filename)  
 except Exception as e:  
 print(f"Error loading pickle data: {e}")  
 return  
 else:  
 print("Invalid choice for loading data.")  
 return  
  
 products = create\_product\_objects(loaded\_data)  
  
 product\_query = input("Enter product name to display export info: ").strip()  
 if product\_query in products:  
 print("\nProduct Information:")  
 print(products[product\_query])  
 else:  
 print("Product not found in the export data.")  
  
 sorted\_products = sort\_products\_by\_total\_volume(loaded\_data)  
 print("\nProducts sorted by total export volume (descending):")  
 for name, volume in sorted\_products:  
 print(f"{name}: {volume}")  
  
 if product\_query in products:  
 countries = products[product\_query].importing\_countries()  
 print(f"\nThe product '{product\_query}' is exported to the following countries: {countries}")  
 print(f"Total export volume: {products[product\_query].total\_volume()}")  
  
 repeat = input("\nDo you want to run Task 1 again? (y/n): ").strip().lower()  
 if repeat == 'y':  
 task1()  
 else:  
 print("Returning to main menu.")

Задание 2. В соответствии с заданием своего варианта составить программу для анализа текста. Считать из исходного файла текст. Используя регулярные выражения получить искомую информацию (см. условие), вывести ее на экран и сохранить в другой файл. Заархивировать файл с результатом с помощью модуля zipfile и обеспечить получение информации о файле в архиве.

Также выполнить общее задание – определить и сохранить в файл с результатами:

* количество предложений в тексте;
* количество предложений в тексте каждого вида отдельно (повествовательные, вопросительные и побудительные);
* среднюю длину предложения в символах (считаются только слова);
* среднюю длину слова в тексте в символах;
* количество смайликов в заданном тексте. Смайликом будем считать последовательность символов, удовлетворяющую условиям:

первым символом является либо «;» (точка с запятой) либо «:» (двоеточие) ровно один раз;

далее может идти символ «-» (минус) сколько угодно раз (в том числе символ минус может идти ноль раз);

в конце обязательно идет некоторое количество (не меньше одной) одинаковых скобок из следующего набора: «(», «)», «[», «]»;

внутри смайлика не может встречаться никаких других символов. Например, эта последовательность является смайликом: «;---------[[[[[[[[». Эти последовательности смайликами не являются: «]», «;--»,«:»,«)».

Вывести все заглавные английские буквы

В заданном тексте заменить последовательность символов «a…ab…bc…c» (букв a и c в последовательности больше 0, букв b – больше единицы) на последовательность «qqq».

Определить, сколько слов имеют максимальную длину;

Вывести все слова, за которыми следует запятая или точка;

Найти самое длинное слово, которое заканчивается на 'е'

import os  
import zipfile  
  
def get\_data\_directory():  
 *"""  
 Returns the path to the 'data' directory within Task2.  
 Creates it if it does not exist.  
 """* current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
 data\_dir = os.path.join(current\_dir, "data")  
 os.makedirs(data\_dir, exist\_ok=True)  
 return data\_dir  
  
def read\_text\_file(filename):  
 *"""  
 Read text from the specified file located in the data directory.  
 Args:  
 filename (str): The filename (e.g., "input\_text.txt").  
 Returns:  
 str: The content of the file.  
 """* data\_dir = get\_data\_directory()  
 full\_path = os.path.join(data\_dir, filename)  
 with open(full\_path, "r", encoding="utf-8") as f:  
 return f.read()  
  
def write\_text\_file(filename, content):  
 *"""  
 Write the provided content to the specified file in the data directory.  
 Args:  
 filename (str): The filename for the output file.  
 content (str): The text content to write.  
 """* data\_dir = get\_data\_directory()  
 full\_path = os.path.join(data\_dir, filename)  
 with open(full\_path, "w", encoding="utf-8") as f:  
 f.write(content)  
  
def archive\_file(file\_to\_archive, zip\_filename):  
 *"""  
 Archive the specified file (located in the data directory) into a zip archive.  
 The resulting zip file is also saved in the data directory.  
 Args:  
 file\_to\_archive (str): Name of the file to be archived (e.g., "result\_text\_analysis.txt").  
 zip\_filename (str): Name for the resulting zip file.  
 """* data\_dir = get\_data\_directory()  
 file\_full\_path = os.path.join(data\_dir, file\_to\_archive)  
 zip\_full\_path = os.path.join(data\_dir, zip\_filename)  
  
 with zipfile.ZipFile(zip\_full\_path, 'w') as zipf:  
 zipf.write(file\_full\_path, arcname=os.path.basename(file\_full\_path))  
  
def get\_archive\_info(zip\_filename):  
 *"""  
 Retrieve information about files contained in the zip archive stored in the data directory.  
 Args:  
 zip\_filename (str): The filename of the zip archive.  
 Returns:  
 str: Information about each file in the archive.  
 """* data\_dir = get\_data\_directory()  
 zip\_full\_path = os.path.join(data\_dir, zip\_filename)  
 info\_str = ""  
 with zipfile.ZipFile(zip\_full\_path, 'r') as zipf:  
 for info in zipf.infolist():  
 info\_str += f"Filename: {info.filename}\n"  
 info\_str += f"File size: {info.file\_size} bytes\n"  
 info\_str += f"Compressed size: {info.compress\_size} bytes\n"  
 info\_str += f"Modified date: {info.date\_time}\n"  
 info\_str += "--------------------------\n"  
 return info\_str

import re  
  
class LoggingMixin:  
 *"""  
 Mixin class providing simple logging functionality.  
 """* def log(self, message):  
 *"""Log a message to the console."""* print(f"[LOG]: {message}")  
  
class BaseTextAnalyzer:  
 *"""  
 Base class for text analysis.  
 Attributes:  
 instance\_count (int): Static attribute to count how many analyzers were created.  
 """* instance\_count = 0  
  
 def \_\_init\_\_(self, text):  
 *"""  
 Initialize the analyzer with provided text.  
 Args:  
 text (str): The text to analyze.  
 """* if not isinstance(text, str):  
 raise ValueError("Text must be a string.")  
 self.\_text = text  
 BaseTextAnalyzer.instance\_count += 1  
  
 @property  
 def text(self):  
 *"""Getter for the text."""* return self.\_text  
  
 @text.setter  
 def text(self, value):  
 *"""  
 Setter for the text.  
 Args:  
 value (str): New text value.  
 Raises:  
 ValueError: If value is not a string.  
 """* if isinstance(value, str):  
 self.\_text = value  
 else:  
 raise ValueError("Text must be a string.")  
  
 def \_\_str\_\_(self):  
 *"""  
 Magic method to return a string representation of the analyzer.  
 Returns:  
 str: Summary info including the length of text.  
 """* return f"Text Analyzer (text length: {len(self.text)})"  
  
class RegexTextAnalyzer(BaseTextAnalyzer, LoggingMixin):  
 *"""  
 Concrete text analyzer that uses regular expressions to extract various information  
 from text.  
 Inherits from BaseTextAnalyzer and LoggingMixin.  
 """* def analyze\_sentences(self):  
 *"""  
 Analyze sentences in the text.  
 The method counts the total number of sentences and the number of sentences by type:  
 - Declarative: ending with '.'  
 - Interrogative: ending with '?'  
 - Exclamatory: ending with '!'  
 Returns:  
 dict: Dictionary with keys 'total', 'declarative', 'interrogative', 'exclamatory'.  
 """* sentence\_pattern = r'[^.!?]+[.!?]'  
 sentences = re.findall(sentence\_pattern, self.text, flags=re.UNICODE)  
 total = len(sentences)  
 declarative = sum(1 for s in sentences if s.strip().endswith('.'))  
 interrogative = sum(1 for s in sentences if s.strip().endswith('?'))  
 exclamatory = sum(1 for s in sentences if s.strip().endswith('!'))  
 self.log(f"Found {total} sentences in total.")  
 return {  
 "total": total,  
 "declarative": declarative,  
 "interrogative": interrogative,  
 "exclamatory": exclamatory  
 }  
  
 def average\_sentence\_length(self):  
 *"""  
 Calculate the average length of sentences (considering only letters in words).  
 For each sentence, only the characters belonging to words (letters and digits) are counted.  
 Returns:  
 float: The average sentence length in characters.  
 """* sentence\_pattern = r'[^.!?]+[.!?]'  
 sentences = re.findall(sentence\_pattern, self.text, flags=re.UNICODE)  
 if not sentences:  
 return 0.0  
 lengths = []  
 for sent in sentences:  
 words = re.findall(r'\b\w+\b', sent, flags=re.UNICODE)  
  
 length = sum(len(word) for word in words)  
 lengths.append(length)  
 avg\_length = sum(lengths) / len(lengths) if lengths else 0.0  
 self.log(f"Average sentence length (letters only): {avg\_length:.2f}")  
 return avg\_length  
  
 def average\_word\_length(self):  
 *"""  
 Calculate the average word length in the entire text.  
 Returns:  
 float: Average number of characters per word.  
 """* words = re.findall(r'\b[a-zA-Zа-яА-ЯёЁ]+\b', self.text, flags=re.UNICODE)  
 if not words:  
 return 0.0  
 avg = sum(len(word) for word in words) / len(words)  
 self.log(f"Average word length: {avg:.2f}")  
 return avg  
  
 def count\_smileys(self):  
 *"""  
 Count the number of smileys in the text.  
 A smiley is defined as:  
 - The first character is ':' or ';' (exactly one occurrence)  
 - Followed by zero or more '-' (minus) symbols  
 - Ending with at least one bracket symbol; all such brackets must be identical  
 and can be one of: '(', ')', '[' or ']'  
 Returns:  
 int: The number of smiley occurrences.  
 """* pattern = r"[:;]-\*([([{)\]}])+"  
 matches = re.findall(pattern, self.text)  
 count = len(matches)  
 self.log(f"Found {count} smileys in the text.")  
 return count  
  
 def get\_uppercase\_letters(self):  
 *"""  
 Extract all uppercase English letters from the text.  
 Returns:  
 list: List of uppercase letters found.  
 """* letters = re.findall(r"[A-Z]", self.text)  
 self.log(f"Found {len(letters)} uppercase English letters.")  
 return letters  
  
 def replace\_pattern(self):  
 *"""  
 Replace in the text all sequences matching the pattern "a…ab…bc…c" with "qqq".  
 The pattern is defined as:  
 - One or more 'a'  
 - Followed by two or more 'b'  
 - Followed by one or more 'c'  
 Returns:  
 str: The modified text after replacement.  
 """* pattern = r"a+b{2,}c+"  
 replaced\_text = re.sub(pattern, "qqq", self.text, flags=re.IGNORECASE)  
 self.log("Replaced all occurrences of pattern a+b{2,}c+ with 'qqq'.")  
 return replaced\_text  
  
 def count\_max\_length\_words(self):  
 *"""  
 Determine the number of words having the maximum length in the text.  
 Returns:  
 int: Count of words whose length equals the maximum word length.  
 """* words = re.findall(r'\b[a-zA-Zа-яА-ЯёЁ]+\b', self.text, flags=re.UNICODE)  
 if not words:  
 return 0  
 max\_length = max(len(word) for word in words)  
 count = sum(1 for word in words if len(word) == max\_length)  
 self.log(f"Maximum word length is {max\_length} and {count} words have this length.")  
 return count  
  
 def get\_words\_followed\_by\_punct(self):  
 *"""  
 Extract all words that are immediately followed by a comma or a period.  
 Returns:  
 list: List of such words.  
 """* words = re.findall(r'\b([a-zA-Zа-яА-ЯёЁ]+)(?=[,.])', self.text, flags=re.UNICODE)  
 self.log(f"Found {len(words)} words followed by a comma or period.")  
 return words  
  
 def longest\_word\_ending\_on\_e(self):  
 *"""  
 Find the longest word in the text that ends with the letter 'е' (case-insensitive).  
 Returns:  
 str: The longest word ending with 'е'. If several words have the same length,  
 the first found is returned. Returns an empty string if none found.  
 """* words = re.findall(r'\b[a-zA-Zа-яА-ЯёЁ]+\b', self.text, flags=re.UNICODE)  
  
 filtered = [word for word in words if re.search(r'е$', word, flags=re.IGNORECASE)]  
 if not filtered:  
 self.log("No word ending with 'е' was found.")  
 return ""  
 longest = max(filtered, key=len)  
 self.log(f"The longest word ending with 'е' is '{longest}'.")  
 return longest

import os  
from .text\_analyzer import RegexTextAnalyzer  
from . import file\_manager  
  
def task2():  
 *"""  
 Execute Task 2: Text Analysis.  
 Steps:  
 1. Read text from input file.  
 2. Analyze text: sentence count (by type), average lengths, smileys count, etc.  
 3. Perform additional regex tasks including extraction and substitution.  
 4. Display and save the results.  
 5. Archive the result file and show archive info.  
 6. Provide an option to re-run the task.  
 """* input\_filename = "input\_text.txt"  
 result\_filename = "result\_text\_analysis.txt"  
 zip\_filename = "result\_archive.zip"  
  
 data\_dir = file\_manager.get\_data\_directory()  
 input\_full\_path = os.path.join(data\_dir, input\_filename)  
  
 if not os.path.exists(input\_full\_path):  
 print(  
 f"Input file '{input\_filename}' not found in {data\_dir}. Please create this file with the text to analyze.")  
 return  
  
 text = file\_manager.read\_text\_file(input\_filename)  
  
 analyzer = RegexTextAnalyzer(text)  
  
 analysis\_results = []  
 analysis\_results.append("------ Text Analysis Results ------")  
  
 sentences\_info = analyzer.analyze\_sentences()  
 analysis\_results.append(f"Total number of sentences: {sentences\_info['total']}")  
 analysis\_results.append(f"Declarative (ending with '.'): {sentences\_info['declarative']}")  
 analysis\_results.append(f"Interrogative (ending with '?'): {sentences\_info['interrogative']}")  
 analysis\_results.append(f"Exclamatory (ending with '!'): {sentences\_info['exclamatory']}")  
  
 avg\_sentence\_len = analyzer.average\_sentence\_length()  
 analysis\_results.append(f"Average sentence length (word characters only): {avg\_sentence\_len:.2f}")  
  
 avg\_word\_len = analyzer.average\_word\_length()  
 analysis\_results.append(f"Average word length: {avg\_word\_len:.2f}")  
  
 smiley\_count = analyzer.count\_smileys()  
 analysis\_results.append(f"Number of smileys: {smiley\_count}")  
  
 uppercase\_letters = analyzer.get\_uppercase\_letters()  
 if uppercase\_letters:  
 analysis\_results.append("Uppercase English letters: " + ", ".join(uppercase\_letters))  
 else:  
 analysis\_results.append("No uppercase English letters found.")  
  
 replaced\_text = analyzer.replace\_pattern()  
 analysis\_results.append("Text after replacing pattern a+b{2,}c+ with 'qqq':")  
 analysis\_results.append(replaced\_text)  
  
 max\_length\_words\_count = analyzer.count\_max\_length\_words()  
 analysis\_results.append(f"Number of words with maximum length: {max\_length\_words\_count}")  
  
 words\_punct = analyzer.get\_words\_followed\_by\_punct()  
 analysis\_results.append("Words followed by a comma or period: " + ", ".join(words\_punct))  
  
 longest\_word = analyzer.longest\_word\_ending\_on\_e()  
 analysis\_results.append(f"Longest word ending with 'е': {longest\_word}")  
  
 result\_text = "\n".join(analysis\_results)  
 print(result\_text)  
  
 file\_manager.write\_text\_file(result\_filename, result\_text)  
 print(f"\nAnalysis results have been saved to '{result\_filename}' in {data\_dir}.")  
  
 try:  
 file\_manager.archive\_file(result\_filename, zip\_filename)  
 print(f"Result file has been archived as '{zip\_filename}' in {data\_dir}.")  
 except Exception as e:  
 print(f"Error archiving result file: {e}")  
 return  
  
 archive\_info = file\_manager.get\_archive\_info(zip\_filename)  
 print("\n---- Archive File Information ----")  
 print(archive\_info)  
  
 repeat = input("Do you want to run Task 2 again? (y/n): ").strip().lower()  
 if repeat == 'y':  
 task2()  
 else:  
 print("Returning to main menu.")

Задание 3. В соответствии с заданием своего варианта доработать программу из ЛР3, использовав класс и обеспечить:

а) определение дополнительных параметров среднее арифметическое элементов последовательности, медиана, мода, дисперсия, СКО последовательности;

б) с помощью библиотеки matplotlib нарисовать графики разных цветов в одной координатной оси:

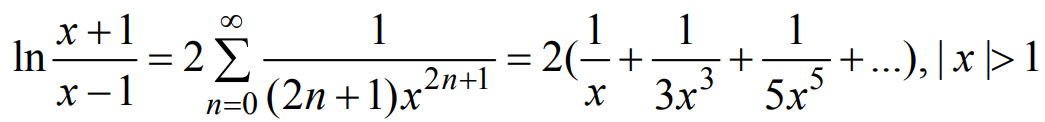
* график по полученным данным разложения функции в ряд, представленным в таблице,
* график соответствующей функции, представленной с помощью модуля math. Обеспечить отображение координатных осей, легенды, текста и аннотации.

Изображение выглядит как снимок экрана, текст, линия, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Здесь x – значение аргумента, F(x) – значение функции, n – количество просуммированных членов ряда, Math F(x) – значение функции, вычисленное с помощью модуля math.

в) сохранить графики в файл.



import math  
import statistics  
  
class SeriesAnalyzer:  
 *"""  
 Class for computing series expansion of F(x) and for calculating statistics.  
 The series approximates:  
 F(x) = ln((x+1)/(x-1))  
 as:  
 F(x) ≈ 2 \* ∑[n=0..∞] 1/((2\*n+1) \* x^(2\*n+1))  
 (for |x| > 1).  
 Attributes:  
 x (float): The argument (|x| > 1).  
 eps (float): Desired accuracy.  
 max\_iterations (int): Maximum allowed iterations.  
 terms (list): List of series terms.  
 result (float): Series approximation F(x).  
 n (int): Number of terms used.  
 mathF (float): Exact F(x) computed with math.log.  
 """* def \_\_init\_\_(self, x, eps, max\_iterations=500):  
 *"""  
 Initializes the SeriesAnalyzer with the argument x and desired precision eps.  
 Args:  
 x (float): Function argument; must satisfy |x| > 1.  
 eps (float): Desired precision; calculation stops when the difference between  
 2\*series\_sum and math.log((x+1)/(x-1)) is less than eps.  
 max\_iterations (int, optional): Maximum number of iterations. Defaults to 500.  
 Raises:  
 ValueError: If |x| is not greater than 1.  
 """* if abs(x) <= 1:  
 raise ValueError("x must satisfy |x| > 1")  
 self.x = x  
 self.eps = eps  
 self.max\_iterations = max\_iterations  
 self.terms = []  
 self.result = None  
 self.n = None  
 self.mathF = None  
 self.calculated = False  
  
 def calculate\_series(self):  
 *"""  
 Compute the series expansion approximation for F(x) = ln((x+1)/(x-1)).  
 Returns:  
 dict: Contains "x", "F(x)" (series approximation),  
 "Math F(x)" (computed with math.log), "n" (number of iterations),  
 and "terms" (list of computed series terms).  
 """* n = 0  
 term = 1 / self.x  
 self.terms.append(term)  
 result = term  
 self.mathF = math.log((self.x + 1) / (self.x - 1))  
  
 while abs(self.mathF - 2 \* result) > self.eps and n < self.max\_iterations:  
 n += 1  
 term = 1 / ((2 \* n + 1) \* (self.x \*\* (2 \* n + 1)))  
 self.terms.append(term)  
 result += term  
  
 self.n = n  
 self.result = 2 \* result  
 self.calculated = True  
  
 return {  
 "x": self.x,  
 "F(x)": self.result,  
 "Math F(x)": self.mathF,  
 "n": self.n,  
 "terms": self.terms  
 }  
  
 def compute\_stats(self):  
 *"""  
 Compute statistical parameters for the series terms:  
 - Arithmetic mean  
 - Median  
 - Mode  
 - Variance  
 - Standard deviation  
 Returns:  
 dict: Keys 'mean', 'median', 'mode', 'variance' and 'std\_dev'.  
 """* if not self.calculated:  
 raise Exception("Series has not been calculated yet. Call calculate\_series() first.")  
  
 if not self.terms:  
 return None  
  
 try:  
 mode\_value = statistics.mode(self.terms)  
 except statistics.StatisticsError:  
 mode\_value = "No unique mode"  
  
 stats\_data = {  
 "mean": statistics.mean(self.terms),  
 "median": statistics.median(self.terms),  
 "mode": mode\_value,  
 "variance": statistics.variance(self.terms) if len(self.terms) > 1 else 0.0,  
 "std\_dev": statistics.stdev(self.terms) if len(self.terms) > 1 else 0.0  
 }  
 return stats\_data

import os  
import matplotlib.pyplot as plt  
from .series\_analyzer import SeriesAnalyzer  
from .utils import get\_data\_directory  
  
class SeriesPlotter:  
 *"""  
 SeriesPlotter computes and plots the series approximation and the math.log-based function F(x)  
 over a range of x values.  
 Methods:  
 compute\_for\_range(x\_values): Computes F(x) via series and math.log for each x.  
 plot(x\_values, series\_vals, math\_vals, filename): Plots the resulting graphs and  
 saves the plot in the data directory.  
 """* def \_\_init\_\_(self, eps, max\_iterations=500):  
 *"""  
 Initialize the SeriesPlotter with given precision and maximum iterations.  
 Args:  
 eps (float): Precision for the series computation.  
 max\_iterations (int, optional): Maximum iterations for the series computation.  
 Defaults to 500.  
 """* self.eps = eps  
 self.max\_iterations = max\_iterations  
  
 def compute\_for\_range(self, x\_values):  
 *"""  
 Computes F(x) values for each x in the provided range using both the series expansion  
 and the math.log function.  
 Args:  
 x\_values (list): List of x values.  
 Returns:  
 tuple: Three lists containing:  
 - series\_vals: The F(x) values computed via the Taylor series.  
 - math\_vals: The F(x) values computed using math.log.  
 - n\_terms: The number of iterations used for each x.  
 """* series\_vals = []  
 math\_vals = []  
 n\_terms = []  
  
 for x in x\_values:  
 try:  
 analyzer = SeriesAnalyzer(x, self.eps, self.max\_iterations)  
 result = analyzer.calculate\_series()  
 series\_vals.append(result["F(x)"])  
 math\_vals.append(result["Math F(x)"])  
 n\_terms.append(result["n"])  
 except Exception as e:  
 print(f"Error computing for x = {x}: {e}")  
 series\_vals.append(None)  
 math\_vals.append(None)  
 n\_terms.append(None)  
 return series\_vals, math\_vals, n\_terms  
  
 def plot(self, x\_values, series\_vals, math\_vals, filename="series\_plot.png"):  
 *"""  
 Plots the series approximation and the math.log-based function on the same coordinate axes.  
 The plot includes:  
 - Two curves with different colors.  
 - Labeled axes.  
 - A legend.  
 - A grid.  
 - An annotation for the point with the maximum difference between the curves.  
 The plot is saved into the 'data' directory within Task3.  
 Args:  
 x\_values (list): x-axis values.  
 series\_vals (list): F(x) computed via the series.  
 math\_vals (list): F(x) computed using math.log.  
 filename (str, optional): File name for saving the plot. Defaults to "series\_plot.png".  
 """* plt.figure(figsize=(10, 8))  
 plt.plot(x\_values, series\_vals, 'bo-', label="Series F(x)")  
 plt.plot(x\_values, math\_vals, 'r\*-', label="Math F(x)")  
 plt.xlabel("x", fontsize=12)  
 plt.ylabel("F(x)", fontsize=12)  
 plt.title("Series Approximation vs Math Function", fontsize=14)  
 plt.legend(fontsize=12)  
 plt.grid(True)  
  
 differences = [abs(s - m) for s, m in zip(series\_vals, math\_vals)  
 if s is not None and m is not None]  
 if differences:  
 max\_diff = max(differences)  
 if max\_diff > 0:  
 idx = differences.index(max\_diff)  
 plt.annotate(f"Max diff: {max\_diff:.2e}",  
 xy=(x\_values[idx], series\_vals[idx]),  
 xytext=(x\_values[idx], series\_vals[idx] + 0.5),  
 arrowprops=dict(facecolor='black', shrink=0.05),  
 fontsize=10)  
  
 data\_dir = get\_data\_directory()  
 file\_path = os.path.join(data\_dir, filename)  
 plt.savefig(file\_path)  
 print(f"Plot saved as {file\_path}")  
 plt.show()

import os  
import random  
  
def input\_float(prompt, min\_val=None):  
 *"""  
 Prompt the user to input a float value and validate it.  
 Args:  
 prompt (str): The prompt message to display to the user.  
 min\_val (float, optional): The minimum required value (if any).  
 Returns:  
 float: The validated float value.  
 """* while True:  
 try:  
 val = float(input(prompt))  
 if min\_val is not None and val <= min\_val:  
 print(f"Value must be greater than {min\_val}.")  
 continue  
 return val  
 except ValueError:  
 print("Invalid input. Please enter a valid float.")  
  
def generate\_value(low, high):  
 *"""  
 Generate a random float value between low and high.  
 Args:  
 low (float): Lower bound.  
 high (float): Upper bound.  
 Returns:  
 float: Randomly generated number.  
 """* return random.uniform(low, high)  
  
def get\_data\_directory():  
 *"""  
 Returns the absolute path to the 'data' directory within Task3.  
 Creates it if it does not exist.  
 """* current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
 data\_dir = os.path.join(current\_dir, "data")  
 os.makedirs(data\_dir, exist\_ok=True)  
 return data\_dir

from prettytable import PrettyTable  
from .series\_analyzer import SeriesAnalyzer  
from .series\_plotter import SeriesPlotter  
from . import utils  
import os  
  
def display\_table(result, stats, eps):  
 *"""  
 Display the series calculation results and basic parameters in a table.  
 Args:  
 result (dict): Dictionary containing series result values.  
 stats (dict): Dictionary containing statistical parameters.  
 eps (float): The precision used in the calculation.  
 """* table = PrettyTable()  
 table.field\_names = ["x", "n", "F(x)", "Math F(x)", "eps"]  
 table.add\_row([  
 result["x"],  
 result["n"],  
 f"{result['F(x)']:.5f}",  
 f"{result['Math F(x)']:.5f}",  
 f"{eps:.1e}"  
 ])  
 print("=== Series Calculation Results ===")  
 print(table)  
  
 stat\_table = PrettyTable()  
 stat\_table.field\_names = ["Statistic", "Value"]  
 for key, value in stats.items():  
 stat\_table.add\_row([key, value])  
 print("=== Statistics of Series Terms ===")  
 print(stat\_table)  
  
def save\_report(report\_text, filename="series\_report.txt"):  
 *"""  
 Save a text report to a file in the data directory.  
 Args:  
 report\_text (str): The report content.  
 filename (str, optional): The report file name. Defaults to "series\_report.txt".  
 """* data\_dir = utils.get\_data\_directory()  
 file\_path = os.path.join(data\_dir, filename)  
 with open(file\_path, "w", encoding="utf-8") as f:  
 f.write(report\_text)  
 print(f"Report saved as {file\_path}")  
  
def task3():  
 *"""  
 Main function for Task 3 interactive menu.  
 Offers the following options:  
 1) Calculate series manually (user inputs x and eps).  
 2) Generate random values for x and eps and calculate series.  
 3) Plot the series approximation and math function for a range of x values.  
 0) Exit the program.  
 """* while True:  
 print("\n== Task 3: Series Analysis and Plotting ==")  
 print("1. Calculate series manually")  
 print("2. Generate values automatically")  
 print("3. Plot series and math function for a range of x values")  
 print("0. Exit from Task 3")  
 choice = input("Choose an option: ").strip()  
  
 if choice == '1':  
 try:  
 x = utils.input\_float("Enter x (|x| > 1, e.g., 1.5): ", 1)  
 eps = utils.input\_float("Enter precision (eps, positive): ", 0)  
 except Exception as e:  
 print(f"Error in input: {e}")  
 continue  
 try:  
 analyzer = SeriesAnalyzer(x, eps)  
 result = analyzer.calculate\_series()  
 stats = analyzer.compute\_stats()  
 display\_table(result, stats, eps)  
  
 report\_text = f"x: {result['x']}\nn: {result['n']}\nSeries F(x): {result['F(x)']:.5f}\nMath F(x): {result['Math F(x)']:.5f}\neps: {eps:.1e}\n"  
 for stat, val in stats.items():  
 report\_text += f"{stat}: {val}\n"  
 save\_report(report\_text)  
 except Exception as e:  
 print(f"An error occurred: {e}")  
 elif choice == '2':  
 x = utils.generate\_value(1.1, 5)  
 eps = utils.generate\_value(1e-5, 1e-2)  
 print(f"Generated values: x = {x:.3f}, eps = {eps:.1e}")  
 try:  
 analyzer = SeriesAnalyzer(x, eps)  
 result = analyzer.calculate\_series()  
 stats = analyzer.compute\_stats()  
 display\_table(result, stats, eps)  
 report\_text = f"x: {result['x']}\nn: {result['n']}\nSeries F(x): {result['F(x)']:.5f}\nMath F(x): {result['Math F(x)']:.5f}\neps: {eps:.1e}\n"  
 for stat, val in stats.items():  
 report\_text += f"{stat}: {val}\n"  
 save\_report(report\_text)  
 except Exception as e:  
 print(f"An error occurred: {e}")  
 elif choice == '3':  
 try:  
 eps = utils.input\_float("Enter precision (eps, positive): ", 0)  
 except Exception as e:  
 print(f"Error in input: {e}")  
 continue  
  
 x\_values = [round(1.1 + i \* 0.2, 3) for i in range(20)]  
 plotter = SeriesPlotter(eps)  
 series\_vals, math\_vals, \_ = plotter.compute\_for\_range(x\_values)  
 plotter.plot(x\_values, series\_vals, math\_vals)  
 elif choice == '0':  
 print("Exiting Task 3.")  
 break  
 else:  
 print("Invalid choice. Please select a valid option.")

Задание 4. В соответствии с заданием своего варианта разработать базовые классы и классы наследники.

Требования по использованию классов:

Абстрактный класс «Геометрическая фигура» содержит абстрактный метод для вычисления площади фигуры

(<https://docs.python.org/3/library/abc.html> )

Класс «Цвет фигуры» содержит свойство для описания цвета геометрической фигуры

(<https://docs.python.org/3/library/functions.html#property> )

Класс «Прямоугольник» (Круг, Ромб, Квадрат, Треугольник и т. д.) наследуется от класса «Геометрическая фигура». Класс должен содержать конструктор по параметрам «ширина», «высота» (для другого типа фигуры соответствующие параметры, например, для круга задаем «радиус») и «цвет». В конструкторе создается объект класса «Цвет фигуры» для хранения цвета. Класс должен переопределять метод, вычисляющий площадь фигуры <https://docs.python.org/3/library/math.html> .

Для класса «Прямоугольник»(тип фигуры в инд. задании).

Определить метод, который возвращает в виде строки основные параметры фигуры, ее цвет и площадь. Использовать метод format (<https://pyformat.info/> ).

Название фигуры должно задаваться в виде поля данных класса и возвращаться методом класса.

В корневом каталоге проекта создайте файл main.py для тестирования классов. Используйте конструкцию, описанную в <https://docs.python.org/3/library/__main__.html>

Пример объекта: Прямоугольник синего цвета шириной 5 и высотой 8.

Программа должна содержать следующие базовые функции:

1. ввод значений параметров пользователем;
2. проверка корректности вводимых данных;
3. построение, закрашивание фигуры в выбранный цвет, введенный с клавиатуры, и подпись фигуры текстом, введенным с клавиатуры;
4. вывод фигуры на экран и в файл.

Построить равнобедренный треугольник с основанием a и высотой h.

class FigureColor:  
 *"""  
 Class for handling the color property of a geometric figure.  
 Attributes:  
 \_color (str): Internal storage for color.  
 """* def \_\_init\_\_(self, color):  
 *"""  
 Initializes a FigureColor object with the provided color.  
 Args:  
 color (str): The color value; must be a non-empty string.  
 Raises:  
 ValueError: If the color is not a valid non-empty string.  
 """* self.\_color = None  
 self.color = color # use setter for validation  
  
 @property  
 def color(self):  
 *"""  
 Getter for the color.  
 Returns:  
 str: The color of the figure.  
 """* return self.\_color  
  
 @color.setter  
 def color(self, value):  
 *"""  
 Setter for the color.  
 Args:  
 value (str): The color to set.  
 Raises:  
 ValueError: If the provided value is not a non-empty string.  
 """* if not isinstance(value, str) or not value:  
 raise ValueError("Color must be a non-empty string.")  
 self.\_color = value  
  
 def \_\_str\_\_(self):  
 *"""  
 String representation of the color.  
 Returns:  
 str: The color.  
 """* return self.color

from abc import ABC, abstractmethod  
  
class GeometricFigure(ABC):  
 *"""  
 Abstract class representing a geometric figure.  
 Attributes:  
 figure\_name (str): Name of the figure (to be defined in subclasses).  
 """* figure\_name = "Figure"  
  
 @abstractmethod  
 def area(self):  
 *"""  
 Abstract method to calculate the area of the geometric figure.  
 Returns:  
 float: Area of the figure.  
 """* pass  
  
 @classmethod  
 def get\_figure\_name(cls):  
 *"""  
 Returns the name of the figure.  
 Returns:  
 str: Figure name.  
 """* return cls.figure\_name

import matplotlib.pyplot as plt  
from matplotlib.patches import Rectangle as MplRectangle, Polygon  
import os  
from .utils import get\_data\_directory  
  
def draw\_rectangle(rect\_obj, rect\_text, save\_filename="rectangle.png"):  
 *"""  
 Draws a rectangle using matplotlib, fills it with its color and annotates it,  
 then saves the drawing to the data directory.  
 Args:  
 rect\_obj (Rectangle): Instance of Rectangle.  
 rec\_text (str): Text of the rectangle.  
 save\_filename (str, optional): File name to save the image. Defaults to "rectangle.png".  
 """* fig, ax = plt.subplots()  
  
 patch = MplRectangle((0, 0), rect\_obj.width, rect\_obj.height,  
 facecolor=rect\_obj.color, edgecolor="black", lw=2)  
 ax.add\_patch(patch)  
 ax.set\_xlim(-1, rect\_obj.width + 1)  
 ax.set\_ylim(-1, rect\_obj.height + 1)  
 ax.set\_aspect('equal', 'box')  
 ax.set\_title(str(rect\_obj))  
  
 ax.text(rect\_obj.width/2, rect\_obj.height/2, str(rect\_text),  
 ha="center", va="center", fontsize=10, color="black")  
  
 data\_dir = get\_data\_directory()  
 file\_path = os.path.join(data\_dir, save\_filename)  
 plt.savefig(file\_path)  
 print(f"Rectangle drawing saved as {file\_path}")  
 plt.show()  
  
def draw\_triangle(triangle\_obj, tri\_text, save\_filename="triangle.png"):  
 *"""  
 Draws an isosceles triangle using matplotlib, fills it with its color and annotates it,  
 then saves the drawing to the data directory.  
 Args:  
 triangle\_obj (IsoscelesTriangle): Instance of IsoscelesTriangle.  
 tri\_text (str): Text of the isosceles triangle.  
 save\_filename (str, optional): File name to save the image. Defaults to "triangle.png".  
 """* fig, ax = plt.subplots()  
  
 points = [(0, 0), (triangle\_obj.base, 0), (triangle\_obj.base/2, triangle\_obj.height)]  
 patch = Polygon(points, closed=True, facecolor=triangle\_obj.color, edgecolor="black", lw=2)  
 ax.add\_patch(patch)  
 ax.set\_xlim(-1, triangle\_obj.base + 1)  
 ax.set\_ylim(-1, triangle\_obj.height + 1)  
 ax.set\_aspect('equal', 'box')  
 ax.set\_title(str(triangle\_obj))  
  
 ax.text(rect\_obj.width/2, rect\_obj.height/2, str(tri\_text),  
 ha="center", va="center", fontsize=10, color="black")  
  
 data\_dir = get\_data\_directory()  
 file\_path = os.path.join(data\_dir, save\_filename)  
 plt.savefig(file\_path)  
 print(f"Triangle drawing saved as {file\_path}")  
 plt.show()

from .geometric\_figure import GeometricFigure  
from .figure\_color import FigureColor  
  
class Rectangle(GeometricFigure):  
 *"""  
 Rectangle class representing a rectangle.  
 Attributes:  
 width (float): Width of the rectangle.  
 height (float): Height of the rectangle.  
 \_color\_obj (FigureColor): Object storing the rectangle's color.  
 """* figure\_name = "Rectangle"  
  
 def \_\_init\_\_(self, width, height, color):  
 *"""  
 Initializes a Rectangle object with given width, height, and color.  
 Args:  
 width (float): Positive width.  
 height (float): Positive height.  
 color (str): Color name.  
 Raises:  
 ValueError: If width or height are not positive numbers.  
 """* if width <= 0 or height <= 0:  
 raise ValueError("Width and height must be positive numbers.")  
 self.width = width  
 self.height = height  
 self.\_color\_obj = FigureColor(color)  
  
 @property  
 def color(self):  
 *"""  
 Returns the color of the rectangle.  
 Returns:  
 str: The rectangle's color.  
 """* return self.\_color\_obj.color  
  
 @color.setter  
 def color(self, value):  
 *"""  
 Sets the rectangle's color.  
 Args:  
 value (str): The new color.  
 """* self.\_color\_obj.color = value  
  
 def area(self):  
 *"""  
 Calculates the area of the rectangle.  
 Returns:  
 float: The area.  
 """* return self.width \* self.height  
  
 def \_\_str\_\_(self):  
 *"""  
 Returns a formatted string with rectangle parameters, color, and area.  
 Returns:  
 str: Formatted description.  
 """* return "Rectangle (width: {w:.2f}, height: {h:.2f}, color: '{c}') has an area of {a:.2f}".format(  
 w=self.width, h=self.height, c=self.color, a=self.area()  
 )

from .geometric\_figure import GeometricFigure  
from .figure\_color import FigureColor  
  
class IsoscelesTriangle(GeometricFigure):  
 *"""  
 Class representing an isosceles triangle with a base and height.  
 Attributes:  
 base (float): The base length.  
 height (float): The height.  
 \_color\_obj (FigureColor): Object storing the triangle's color.  
 """* figure\_name = "Isosceles Triangle"  
  
 def \_\_init\_\_(self, base, height, color):  
 *"""  
 Initializes an isosceles triangle.  
 Args:  
 base (float): Positive base length.  
 height (float): Positive height.  
 color (str): Color of the triangle.  
 Raises:  
 ValueError: If base or height are not positive.  
 """* if base <= 0 or height <= 0:  
 raise ValueError("Base and height must be positive numbers.")  
 self.base = base  
 self.height = height  
 self.\_color\_obj = FigureColor(color)  
  
 @property  
 def color(self):  
 *"""  
 Returns the color of the triangle.  
 Returns:  
 str: The triangle's color.  
 """* return self.\_color\_obj.color  
  
 @color.setter  
 def color(self, value):  
 *"""  
 Sets the triangle's color.  
 Args:  
 value (str): The new color.  
 """* self.\_color\_obj.color = value  
  
 def area(self):  
 *"""  
 Calculates the area of the triangle.  
 Returns:  
 float: The area.  
 """* return (self.base \* self.height) / 2  
  
 def \_\_str\_\_(self):  
 *"""  
 Returns a formatted string with triangle parameters, color, and area.  
 Returns:  
 str: Formatted description.  
 """* return "Isosceles Triangle (base: {b:.2f}, height: {h:.2f}, color: '{c}') has an area of {a:.2f}".format(  
 b=self.base, h=self.height, c=self.color, a=self.area()  
 )

import os  
  
def input\_float(prompt, min\_val=None):  
 *"""  
 Prompts the user to enter a float value and validates the input.  
 Args:  
 prompt (str): The message shown to the user.  
 min\_val (float, optional): The minimum acceptable value. Defaults to None.  
 Returns:  
 float: The validated float value.  
 """* while True:  
 try:  
 value = float(input(prompt))  
 if min\_val is not None and value <= min\_val:  
 print(f"Value must be greater than {min\_val}.")  
 continue  
 return value  
 except ValueError:  
 print("Invalid input. Please enter a valid number.")  
  
def get\_data\_directory():  
 *"""  
 Returns the absolute path to the 'data' directory within Task4.  
 If the directory does not exist, it is created.  
 Returns:  
 str: Path to the 'data' directory.  
 """* current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
 data\_dir = os.path.join(current\_dir, "data")  
 os.makedirs(data\_dir, exist\_ok=True)  
 return data\_dir

from .rectangle import Rectangle  
from .triangle import IsoscelesTriangle  
from .drawing import draw\_rectangle, draw\_triangle  
from .utils import input\_float  
  
def task4():  
 *"""  
 Main function for Task 4 interactive menu.  
 Options:  
 1. Create and draw a Rectangle.  
 2. Create and draw an Isosceles Triangle.  
 0. Exit.  
 """* while True:  
 print("\n== Task 4: Geometric Figures ==")  
 print("1. Create and draw a Rectangle")  
 print("2. Create and draw an Isosceles Triangle")  
 print("0. Exit")  
 choice = input("Choose an option: ").strip()  
  
 if choice == '1':  
 try:  
 print("Enter parameters for the Rectangle:")  
 width = input\_float("Width: ", 0)  
 height = input\_float("Height: ", 0)  
 color = input("Enter color: ").strip()  
 rec\_text = input("Enter text: ")  
 rect = Rectangle(width, height, color)  
 print(rect)  
 draw\_rectangle(rect, rec\_text)  
 except Exception as e:  
 print(f"Error: {e}")  
 elif choice == '2':  
 try:  
 print("Enter parameters for the Isosceles Triangle:")  
 base = input\_float("Base: ", 0)  
 height = input\_float("Height: ", 0)  
 color = input("Enter color: ").strip()  
 tri\_text = input("Enter text: ")  
 triangle = IsoscelesTriangle(base, height, color)  
 print(triangle)  
 draw\_triangle(triangle, tri\_text)  
 except Exception as e:  
 print(f"Error: {e}")  
 elif choice == '0':  
 print("Exiting Task 4.")  
 break  
 else:  
 print("Invalid choice. Please try again.")

Задание 5. В соответствии с заданием своего варианта исследовать возможности библиотека NumPy при работе с массивами и математическими и статическими операциями. Сформировать целочисленную матрицу А[n,m] с помощью генератора случайных чисел (random).

а) Библиотека NumPy.

1. Создание массива. Функции array() и values().

2. Функции создания массива заданного вида.

3. Индексирование массивов NumPy. Индекс и срез.

4. Операции с массивами. Универсальные (поэлементные) функции.

б) Математические и статистические операции.

1. Функция mean()

2. Функция median()

3. Функция corrcoef()

4. Дисперсия var().

5. Стандартное отклонение std()

Найти столбец с наименьшей суммой элементов.

Вычислить значение медианы этого столбца. Вычисление медианы выполнить двумя способами: через стандартную функцию и через программирование формулы.

import numpy as np  
import os  
from .utils import get\_data\_directory  
  
def save\_matrix(filename, matrix):  
 *"""  
 Saves a NumPy matrix to a file in the data directory.  
 Args:  
 filename (str): Name of the file to save.  
 matrix (np.ndarray): The matrix to save.  
 """* data\_dir = get\_data\_directory()  
 file\_path = os.path.join(data\_dir, filename)  
 np.savetxt(file\_path, matrix, fmt='%d')  
 print(f"Matrix saved to {file\_path}")  
  
def load\_matrix(filename):  
 *"""  
 Loads a NumPy matrix from a file in the data directory.  
 Args:  
 filename (str): Name of the file to load.  
 Returns:  
 np.ndarray: The loaded matrix.  
 """* data\_dir = get\_data\_directory()  
 file\_path = os.path.join(data\_dir, filename)  
 if not os.path.exists(file\_path):  
 raise FileNotFoundError(f"File {file\_path} not found.")  
 return np.loadtxt(file\_path, dtype=int)

import numpy as np  
  
def create\_random\_matrix(n, m, low=0, high=100):  
 *"""  
 Creates a random integer matrix of shape (n, m).  
 Args:  
 n (int): Number of rows.  
 m (int): Number of columns.  
 low (int, optional): Minimum value for random integers. Defaults to 0.  
 high (int, optional): Maximum value for random integers. Defaults to 100.  
 Returns:  
 np.ndarray: The generated matrix.  
 """* return np.random.randint(low, high, size=(n, m))  
  
def find\_column\_with\_min\_sum(matrix):  
 *"""  
 Finds the index of the column with the minimum sum of elements.  
 Args:  
 matrix (np.ndarray): The matrix.  
 Returns:  
 tuple: (int, np.ndarray) The column index and the column values.  
 """* col\_sums = matrix.sum(axis=0)  
 min\_col\_idx = np.argmin(col\_sums)  
 return min\_col\_idx, matrix[:, min\_col\_idx]

import numpy as np  
  
def calculate\_mean(matrix):  
 *"""  
 Calculates the mean value of the matrix.  
 Args:  
 matrix (np.ndarray): The matrix.  
 Returns:  
 float: The mean value.  
 """* return np.mean(matrix)  
  
def calculate\_median(matrix):  
 *"""  
 Calculates the median value of the matrix.  
 Args:  
 matrix (np.ndarray): The matrix.  
 Returns:  
 float: The median value.  
 """* return np.median(matrix)  
  
def calculate\_column\_median(column):  
 *"""  
 Calculates the median value of a column manually (without using np.median).  
 Args:  
 column (np.ndarray): The column.  
 Returns:  
 float: The median value.  
 """* sorted\_col = np.sort(column)  
 mid = len(sorted\_col) // 2  
 if len(sorted\_col) % 2 == 0:  
 return (sorted\_col[mid - 1] + sorted\_col[mid]) / 2  
 else:  
 return sorted\_col[mid]  
  
def calculate\_variance(matrix):  
 *"""  
 Calculates the variance of the matrix.  
 Args:  
 matrix (np.ndarray): The matrix.  
 Returns:  
 float: The variance.  
 """* return np.var(matrix)  
  
def calculate\_std\_dev(matrix):  
 *"""  
 Calculates the standard deviation of the matrix.  
 Args:  
 matrix (np.ndarray): The matrix.  
 Returns:  
 float: The standard deviation.  
 """* return np.std(matrix)  
  
def calculate\_corrcoef(matrix):  
 *"""  
 Calculates the Pearson correlation coefficients of the matrix.  
 Args:  
 matrix (np.ndarray): The matrix.  
 Returns:  
 np.ndarray: The correlation coefficient matrix.  
 """* return np.corrcoef(matrix, rowvar=False)

import os  
  
def input\_int(prompt, min\_val=None):  
 *"""  
 Prompts the user to enter an integer and validates the input.  
 Args:  
 prompt (str): The message shown to the user.  
 min\_val (int, optional): The minimum acceptable value. Defaults to None.  
 Returns:  
 int: The validated integer value.  
 """* while True:  
 try:  
 value = int(input(prompt))  
 if min\_val is not None and value < min\_val:  
 print(f"Value must be at least {min\_val}.")  
 continue  
 return value  
 except ValueError:  
 print("Invalid input. Please enter a valid integer.")  
  
def get\_data\_directory():  
 *"""  
 Returns the absolute path to the 'data' directory within Task5.  
 If the directory does not exist, it is created.  
 Returns:  
 str: Path to the 'data' directory.  
 """* current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))  
 data\_dir = os.path.join(current\_dir, "data")  
 os.makedirs(data\_dir, exist\_ok=True)  
 return data\_dir

from .utils import input\_int  
from .io\_operations import save\_matrix, load\_matrix  
from .matrix\_operations import create\_random\_matrix, find\_column\_with\_min\_sum  
from .statistics import (calculate\_mean, calculate\_median, calculate\_variance,  
 calculate\_std\_dev, calculate\_corrcoef, calculate\_column\_median)  
  
def task5():  
 *"""  
 Main function for Task 5 interactive menu.  
 Options:  
 1. Create a random matrix.  
 2. Perform statistical operations on the current matrix.  
 3. Find the column with the minimum sum and compute its median.  
 4. Save the current matrix to a file.  
 5. Load a matrix from a file.  
 0. Exit the program.  
 """* matrix = None  
 while True:  
 print("\n== Task 5: NumPy and Matrix Operations ==")  
 print("1. Create a random matrix")  
 print("2. Perform statistical operations on the current matrix")  
 print("3. Find column with minimum sum and compute its median")  
 print("4. Save current matrix to a file")  
 print("5. Load matrix from a file")  
 print("0. Exit")  
 choice = input("Choose an option: ").strip()  
  
 if choice == '1':  
 n = input\_int("Enter number of rows (n): ", 1)  
 m = input\_int("Enter number of columns (m): ", 1)  
 matrix = create\_random\_matrix(n, m)  
 print("Matrix created:")  
 print(matrix)  
 elif choice == '2':  
 if matrix is None:  
 print("Matrix not created yet. Please create a matrix first.")  
 else:  
 mean\_val = calculate\_mean(matrix)  
 median\_val = calculate\_median(matrix)  
 var\_val = calculate\_variance(matrix)  
 std\_dev\_val = calculate\_std\_dev(matrix)  
 corrcoef\_mat = calculate\_corrcoef(matrix)  
 print("Statistical Operations:")  
 print(f"Mean: {mean\_val:.2f}")  
 print(f"Median: {median\_val:.2f}")  
 print(f"Variance: {var\_val:.2f}")  
 print(f"Standard Deviation: {std\_dev\_val:.2f}")  
 print("Correlation Coefficient matrix:")  
 print(corrcoef\_mat)  
 elif choice == '3':  
 if matrix is None:  
 print("Matrix not created yet. Please create a matrix first.")  
 else:  
 col\_idx, col\_vals = find\_column\_with\_min\_sum(matrix)  
 median\_func = calculate\_median(col\_vals)  
 median\_manual = calculate\_column\_median(col\_vals)  
 print(f"Column with minimum sum is at index {col\_idx}")  
 print("Column values:")  
 print(col\_vals)  
 print(f"Median (using np.median): {median\_func:.2f}")  
 print(f"Median (manual calculation): {median\_manual:.2f}")  
 elif choice == '4':  
 if matrix is None:  
 print("Matrix not created yet. Please create a matrix first.")  
 else:  
 filename = input("Enter filename to save (e.g., matrix.txt): ").strip()  
 save\_matrix(filename, matrix)  
 elif choice == '5':  
 filename = input("Enter filename to load (e.g., matrix.txt): ").strip()  
 try:  
 matrix = load\_matrix(filename)  
 print("Matrix loaded:")  
 print(matrix)  
 except Exception as e:  
 print(f"Error loading matrix: {e}")  
 elif choice == '0':  
 print("Exiting Task 5.")  
 break  
 else:  
 print("Invalid option. Please choose a valid option (0-5).")