Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Избранные главы информатики

ОТЧЁТ

к лабораторной работе 4

на тему

Работа с файлами, классами, сериализаторами, регулярными выражениями и стандартными библиотеками

Выполнил: студент группы 253501
Борисевич Александр Михайлович

Проверила: Жвакина Анна Васильевна

Минск 2024

Оглавление

Вариант 4

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.

Ход работы

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

   Модуль управления ходом выполнения задания:

```python
class Task:
    def __init__(self):
        pass


    def pick_serializer(self) -> dflt.Serializer:
        """Prompts the user to select a serializer and returns the selected serializer.

        Returns:
            The selected serializer.
        """
        while True:
            choice = input('Pick Serializer: 1 - CSV; 2 - Pickle: ')

            if choice in ['1', '2']:
                return CSVSerializer() if choice == '1' else PickleSerializer()

            print('Wrong Input, Try Again')


    def search_by_century(self, src: list[dict], century: int) -> list:
        """Searches a list of events by century.

        Args:
            src (list[dict]): The list of events.
            century (int): The century to search for.

        Returns:
            list: The list of events that match the specified century.
        """
        return sorted([event for event in src if str(event['century']) == str(century)],
                      key=lambda item: item['century'])
```

```python
    def dictionary_of_events(self):
        """Creates a dictionary of events from the main_events list.

        Returns:
            dict: A dictionary of events, where the key is the event name and the value is the century.
        """
        src = HistoryEvent.main_events()
        events = []
        for item in src:
            events.append({ "name": item.name, "century": item.century })

        return events


    def execute(self):
        """Executes the task.

        This includes prompting the user to select a serializer, serializing the dictionary of events, and searching for events by century if the user requests it.
        """
        serializer = self.pick_serializer()

        serializer.serialize(self.dictionary_of_events())
        ds = list(serializer.deserialize())

        choice = input('Do You Want To Search Events By Century? (y/n): ').lower()

        if choice in ['y', 'yes']:
            request = InputService.input_specified_type(int,
                                "What Century Do You Want To Search: ")
            response = self.search_by_century(ds, request)

            if len(response) == 0:
                print('Events Are Not Found')
            else:
                print("--  ", end='')
                print('\n--  '.join([item["name"] for item in response]))
```

```python
class HistoryEvent:
    """
    A class to represent a historical event.

    Attributes:
        name (str): The name of the event.
        century (int): The century in which the event occurred.

    Methods:
        __init__(self, name: str, century: int): Initializes a new instance of the HistoryEvent class.
        name(self): Gets the name of the event.
        century(self): Gets the century in which the event occurred.
        name.setter(self, name: str): Sets the name of the event.
        century.setter(self, century: int): Sets the century in which the event occurred.
        main_events(self): Returns a list of main events in the history of Belarus.
    """

    def __init__(self, name: str, century: int):
        self.__name = name
        self.__century = century

    @property
    def name(self):
        """
        Gets the name of the event.

        Returns:
            str: The name of the event.
        """
        return self.__name

    @property
    def century(self):
        """
        Gets the century in which the event occurred.

        Returns:
            int: The century in which the event occurred.
        """
        return self.__century
```

```python
    @property
    def century(self):
        """
        Gets the century in which the event occurred.

        Returns:
            int: The century in which the event occurred.
        """
        return self.__century

    @name.setter
    def name(self, name: str):
        """
        Sets the name of the event.

        Args:
            name (str): The new name of the event.
        """
        self.__name = name

    @century.setter
    def century(self, century: int):
        """
        Sets the century in which the event occurred.

        Args:
            century (int): The new century in which the event occurred.
        """
        self.__century = century

    @staticmethod
    def main_events():
        """
        Returns a list of main events in the history of Belarus.

        Returns:
            list: A list of main events in the history of Belarus.
        """
        events = [
            HistoryEvent("First mention of the towns Turov and Polotsk", 9),
            HistoryEvent("Foundation of the Kievan Rus", 10),
```

```python
class CSVSerializer(Serializer):
    def __init__(self):
        """
        Initializes the CSVSerializer class.
        """
        self.__fname = super()._default_file_name

    def serialize(self, src) -> None:
        """
        Serializes a list of dictionaries into a CSV file.

        Args:
            src (list[dict]): List of dictionaries to be serialized.

        Returns:
            None
        """
        with open(self.__fname, 'w', newline='') as file:
            fieldnames = ['name', 'century']
            writer = csv.DictWriter(file, fieldnames=fieldnames, dialect=csv.unix_dialect)
            writer.writeheader()
            writer.writerows(src)

    def deserialize(self):
        """
        Deserializes data from a CSV file and yields each item as a dictionary.

        Yields:
            dict: A dictionary containing data from the CSV file.
        """
        with open(self.__fname, 'r', newline='') as file:
            reader = csv.DictReader(file)

            for item in reader:
                yield item

```

```python
class PickleSerializer(Serializer):
    """
    A serializer that handles the serialization and deserialization of Python objects using pickle.

    Attributes:
        __fname (str): Filename where the serialized object will be stored. Defaults to the filename specified in the parent class.
    """

    def __init__(self):
        """
        Initializes the pickle_serializer instance, setting the filename to the default file name defined in the parent class.
        """
        self.__fname = super()._default_file_name

    def serialize(self, src: dict) -> None:
        """
        Serializes the given source dictionary into a file using pickle.

        Args:
            src (dict): The dictionary to serialize.

        Returns:
            None
        """
        with open(self.__fname, 'wb') as file:
            pickle.dump(src, file)

    def deserialize(self):
        """
        Deserializes the contents of the file back into a Python object.

        Yields:
            The deserialized Python objects from the file.
        """
        with open(self.__fname, 'rb') as file:
            items = pickle.load(file)

        for item in items:
            yield item
```

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: «]», «;--»,«:»,«)».

Модуль управления ходом выполнения задания:

```python
class Task(TextServiceMixin):
    """
    A class that encapsulates the functionality for reading, processing, and zipping text data.

    Attributes:
        __filepath (str): Path to the input text file.
        __output_filepath (str): Path to the output text file where results will be written.
    """

    def __init__(self, path: str = '/home/main/igilab/l4/second/data/data.txt',
                 output_path='/home/main/igilab/l4/second/data/results.txt'):
        """
        Initializes the Task instance with paths for input and output files.

        Args:
            path (str): The file path for reading data.
            output_path (str): The file path for writing results.
        """
        self.__filepath = path
        self.__output_filepath = output_path

    def read_data_from_file(self) -> str:
        """
        Reads data from the file at the specified file path.

        Returns:
            str: The content of the file as a string.
        """
        with open(self.__filepath, 'r') as text:
            return text.read()

    def zip_results(self):
        """
```

```python
    def zip_results(self):
        """
        Zips the results file and prints the details of the zipped items. Removes the original results file after zipping.
        """
        with ZipFile('/home/main/igilab/l4/second/data/results.zip', 'w',
                     compression=ZIP_DEFLATED, compresslevel=3) as zp:
            zp.write(self.__output_filepath, arcname='results.txt')

            for item in zp.infolist():
                print(f"Filename: {item.filename}, Date: {item.date_time}, Size: {item.file_size}")

        os.remove(self.__output_filepath)

    def execute(self):
        """
        Executes the main functionality of the Task class. It reads data, processes it to calculate various statistics,
        writes the results to an output file, and then zips this output file.
        """
        data = self.read_data_from_file()

        amount_of_dot_sentences = self.amount_of_sentences_by_ending_symbol(data, '.')
        amount_of_question_sentences = self.amount_of_sentences_by_ending_symbol(data, '?')
        amount_of_exclaim_sentences = self.amount_of_sentences_by_ending_symbol(data, '!')
        amount_of_sentences = amount_of_dot_sentences + amount_of_question_sentences + amount_of_exclaim_sentences

        with open(self.__output_filepath, 'w') as output:
            print(f'Amount Of Sentences Is { amount_of_sentences }', file=output)
            print(f'Amount Of Sentences Ending With "." Is { amount_of_dot_sentences }.', file=output)
            print(f'Amount Of Sentences Ending With "?" Is { amount_of_question_sentences }.', file=output)
            print(f'Amount Of Sentences Ending With "!" Is { amount_of_exclaim_sentences }.', file=output)
            print(file=output)

            sentences = self.list_of_sentences(data)
            words = self.list_of_words(sentences)

            print(f'Average Sentence Length Is { self.average_sentence_length(sentences) }.', file=output)
            print(f'Average Word Length Is { self.average_word_length(words) }.', file=output)

            print(f'Amount Of Smiles Is { self.amount_of_smiles(data) }.', file=output)
            print(file=output)
```

```python
class TextServiceMixin:
    """
    A mixin class providing various text processing services such as sentence splitting,
    sentence and word analysis, and pattern matching within texts.
    """

    def list_of_sentences(self, text: str) -> list[str]:
        """
        Splits the given text into a list of sentences.

        Args:
            text (str): The text to split into sentences.

        Returns:
            list[str]: A list of sentences without newline characters and empty entries.
        """
        return [item.replace('\n', '') for item in re.split("[\.\?\!]", text) if len(item) > 0]

    def amount_of_sentences_by_ending_symbol(self, text: str, symbol) -> list[str]:
        """
        Counts the number of sentences ending with a specific symbol.

        Args:
            text (str): The text to analyze.
            symbol (str): The sentence-ending punctuation to count.

        Returns:
            list[str]: The count of sentences ending with the specified symbol.
        """
        return len(re.split(f'\{symbol}', text)) - 1 if text.count(symbol) != 0 else 0

```

```python
    def average_sentence_length(self, sentences: list[str]) -> float:
        """
        Calculates the average length of sentences in a list.

        Args:
            sentences (list[str]): A list of sentences.

        Returns:
            float: The average length of the sentences.
        """
        return sum([len(self.remove_non_letter_symbols(item).replace(' ', '')) for item in sentences]) / len(sentences)

    def list_of_words(self, sentences: list[str]) -> list[str]:
        """
        Creates a list of words from a list of sentences.

        Args:
            sentences (list[str]): A list of sentences.

        Returns:
            list[str]: A list of words.
        """
        return [item for item in list(itertools.chain.from_iterable([self.remove_non_letter_symbols(item).split(' ') for item in sentences])) if item != '']

    def average_word_length(self, words: list[str]) -> float:
        """
        Calculates the average length of words in a list.

        Args:
            words (list[str]): A list of words.

        Returns:
            float: The average length of the words.
        """
        return sum(map(len, words)) / len(words)

```

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.

x	*n*	*F*(*x*)	*Math F*(*x*)	*eps*

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

Модуль управления ходом выполнения задания:

```python
class Task:
    """
    A class designed to execute a sequence of calculations and display statistical information.
    """

    def __init__(self):
        """
        Initializes the Task instance. Currently, the initializer does not perform any actions.
        """
        pass

    def execute(self):
        """
        Executes the main functionality of the Task class. It prompts the user for an accuracy level,
        performs a series of evaluations, and then prints and plots the statistical results.
        """
        # Prompt the user to enter the desired accuracy for calculations.
        eps = InputService.input_specified_type(float, 'Enter The Accuracy: ')

        # Initialize the calculator with the specified accuracy.
        calc = Calculator(eps)

        # List to store the results of the evaluations.
        iterations = []

        # Evaluate the function provided by the calculator for values from -0.99 to 0.99.
        for x in range(-99, 100):
            iterations.append(calc.evaluate(x / 100))

        # Calculate and retrieve statistics for the sequence of evaluations.
        statistics = SequenceStatistics(iterations).stats()

        # Print the statistical results in a formatted manner.
        print('Stats Of Sequence: ')
        print("--  ", end='')
        print('\n--  '.join([f'{ key }: { value }' for key, value in statistics.items()]))

        # Plot the sequence of evaluations using the plot service.
        PlotService.plot_sequence(iterations)
```

```
4   class Calculator:
20      def f(self, x):
22          Computes the function value for a given x.
23
24          Parameters:
25              x (float): Input value.
26
27          Returns:
28              float: The result of mt.log(1 - x).
29          """
30          return mt.log(1 - x)
31
32      def evaluate(self, x):
33          """
34          Evaluates the function using an iterative method.
35
36          Parameters:
37              x (float): Input value.
38
39          Returns:
40              iteration(x, n, result, math_result, self.__eps): An object containing iteration details.
41
42          Raises:
43              ValueError: If the absolute value of x exceeds 1.
44          """
45          if abs(x) >= 1:
46              raise ValueError("Absolute Value Of 'x' Must Be Lower Or Equal To 1")
47
48          n = 1
49          step = -x
50          result = -x
51          math_result = self.f(x)
52
53          for n in range(1, 501):
54              if abs(math_result - result) <= self.__eps:
55                  return Iteration(x, n, result, math_result, self.__eps)
56
57              step *= x * n
58              step /= n + 1
59              result += step
60
61          return Iteration(x, n, result, math_result, self.__eps)
62
```

```python
class PlotService:
    def __init__(self):
        """
        Initializes an instance of the plot_service class.
        """
        pass

    @staticmethod
    def plot_sequence(iterations: list[Iteration]):
        """
        Plots the function values by sequence and math library.

        Parameters:
            iterations (list[Iteration]): A list of Iteration objects containing details for each iteration.

        Returns:
            None

        Saves:
            A plot in 'plot.pdf' file.

        Prints:
            'Plot Is Saved In plot.pdf'
        """
        x_points = [item.x for item in iterations]
        y_points = [item.f for item in iterations]
        math_y_points = [item.math_f for item in iterations]

        mpl.plot(x_points, y_points, color='r', label='By Sequence')
        mpl.plot(x_points, math_y_points, color='g', label='By Math Lib')

        mpl.xlabel('x')
        mpl.ylabel('ln(1 - x)')

        mpl.title('Function ln(1 - x)')
        mpl.legend()
        mpl.savefig('/home/main/igilab/l4/third/plot.pdf')

        print('Plot Is Saved In plot.pdf')

        mpl.show()
```

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:

   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.

Модуль управления ходом выполнения задания:

```python
class Task:
    def __init__(self):
        """
        Initializes an instance of the Task class.
        """
        pass

    def __input_params(self):
        """
        Collects input parameters for creating a triangle.

        Returns:
            tuple: A tuple containing the following values:
                - a (float): Length of side a.
                - b (float): Length of side b.
                - c (float): Value of the angle between these sides.
                - clr (str): Color of the triangle.
                - name (str): Name of the figure.
        """
        a = InputService.input_specified_type(float, 'Input Length Of Side a: ')
        b = InputService.input_specified_type(float, 'Input Length Of Side b: ')
        c = InputService.input_specified_type(float, 'Input Value Of Angle Between These Sides: ')

        print(f'Available Colors: {Color.Colors.keys()}')
        clr = input('Pick One Of Them: ')

        name = input('Enter Name Of Your Figure: ')

        return a, b, c, clr, name
```

```python
    def execute(self):
        """
        Executes the task to create and draw a triangle.

        Returns:
            None
        """
        drawer = drwr()
        a, b, c, clr, name = self.__input_params()

        try:
            tr = Triangle(a, b, c, clr)
            tr.name = name
        except ValueError as ve:
            print(ve)
            print('Using Default Triangle')
            tr = Triangle()
            tr.name = name

        print(tr)
        drawer.draw(tr)
```

Дерево классов для работы с параллелограммом:

```
class Triangle(Figure):
    Attributes:
        __a (float): Length of side a.
        __b (float): Length of side b.
        __c (float): Value of the angle between these sides (in degrees).
        __color (Color): Color of the triangle.
        __name (str): Name of the figure.

    Methods:
        - area(): Calculates the area of the triangle.
        - get_color(): Gets the color of the triangle.
        - available_colors(): Returns available color options.
        - points(): Returns the coordinates of the triangle's vertices.
    """

    def __init__(self, a=3, b=4, c=90, color='r'):
        """
        Initializes an instance of the Triangle class.

        Parameters:
            a (float): Length of side a.
            b (float): Length of side b.
            c (float): Value of the angle between these sides (in degrees).
            color (str): Color of the triangle.

        Raises:
            ValueError: If side lengths or angle are invalid.
        """
        if not color in clr.Colors.keys():
            raise ValueError('Unknown Color')

        if a <= 0 or b <= 0:
            raise ValueError('Length Of Side Must Be Positive')

        if c <= 0 or c >= 180:
            raise ValueError('Angle Must Be In 0..180')

        self.__a = a
        self.__b = b
        self.__c = c
        self.__color = clr(color)
        self.__name = ''
```

```
80      def __str__(self):
81          """
82          Returns a string representation of the triangle.
83
84          Returns:
85              str: A formatted string describing the triangle.
86          """
87          return 'Triangle; a={}, b={}, c={}, area={}.'.format(
88              self.__a, self.__b, self.__c, self.area())
89
90      def get_color(self):
91          """
92          Gets the color of the triangle.
93
94          Returns:
95              str: The color of the triangle.
96          """
97          return self.__color.color
98
99      def available_colors(self):
100         """
101         Returns available color options.
102
103         Returns:
104             list: A list of available color names.
105         """
106         return clr.colors().keys()
107
108     def __rad(self, vl: float):
109         """
110         Converts degrees to radians.
111
112         Parameters:
113             vl (float): Angle in degrees.
114
115         Returns:
116             float: Angle in radians.
117         """
118         return mt.pi * vl / 180.0
119
```

5. В соответствии с заданием своего варианта исследовать возможности библиотека NumPy при работе с массивами и математическими и статическими операциями. Сформировать целочисленную матрицу A[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()

   Модуль управления ходом выполнения задания:

```python
class Task:
    """
    Represents a task that generates a random array and computes statistics.

    Methods:
        - __dimensions(): Collects input dimensions for the array.
        - execute(): Executes the task by generating the array and computing statistics.
    """

    def __init__(self):
        """
        Initializes an instance of the Task class.
        """
        pass

    def __dimensions(self):
        """
        Collects input parameters for the array dimensions.

        Returns:
            tuple: A tuple containing the following values:
                - n (int): Number of rows.
                - m (int): Number of columns.
        """
        n = InputService.input_specified_type(int, 'Enter Amount Of Rows: ')
        m = InputService.input_specified_type(int, 'Enter Amount Of Columns: ')

        return n, m

    def execute(self):
        """
        Executes the task by generating a random array and computing statistics.

        Returns:
            None
        """
        n, m = self.__dimensions()

        asv = ArrayService()
        asv.set_array(ArrayService.random_array(n, m))
        print('Generated Array:')
        print(asv.get_array())
```

Модуль для работы со списком:

```
class ArrayService:

    def get_array(self):
        """
        Get the current array.

        Returns:
            ndarray: The current array.
        """
        return self.__arr

    def set_array(self, arr):
        """
        Set the array.

        Parameters:
            arr (ndarray): The array to set.
        """
        self.__arr = arr

    @staticmethod
    def random_array(n: int, m: int):
        """
        Generate a random array of shape (n, m).

        Parameters:
            n (int): Number of rows.
            m (int): Number of columns.

        Returns:
            ndarray: A random array.
        """
        return np.random.randint(100, size=(n, m))
```

```
3   class ArrayService:
25
26      def get_array(self):
27          """
28          Get the current array.
29
30          Returns:
31              ndarray: The current array.
32          """
33          return self.__arr
34
35      def set_array(self, arr):
36          """
37          Set the array.
38
39          Parameters:
40              arr (ndarray): The array to set.
41          """
42          self.__arr = arr
43
44      @staticmethod
45      def random_array(n: int, m: int):
46          """
47          Generate a random array of shape (n, m).
48
49          Parameters:
50              n (int): Number of rows.
51              m (int): Number of columns.
52
53          Returns:
54              ndarray: A random array.
55          """
56          return np.random.randint(100, size=(n, m))
```

Выводы

В ходе выполнения данной лабораторной работы был освоен базовый синтаксис языка Python, были приобретены навыки работы с файлами, классами, сериализаторами, регулярными выражениями и стандартными библиотеками и закреплены на примере разработки интерактивных приложений.