Лабораторная работа №2

Общее описание

В ходе выполнения лабораторной работы студенту необходимо реализовать сериализатор. Получившийся сериализатор должен корректно сериализовывать (сохранять / упаковывать) и десериализовать (восстанавливать / распаковывать) хранимую информацию. И разработать на основе сериализатора консольную утилиту.

Код вашей программы должен содержать фабричный метод create_serializer(), который будет порождать различные типы сериализаторов: JSON, YAML, TOML, PICKLE. Должна быть возможность легко добавить новый сериализатор, не изменяя архитектуру приложения.

Каждый из сериализаторов должен реализовывать следующие методы:

- dump(obj, fp) — сериализует Python объект в файл
- dumps(obj) — сериализует Python объект в строку
- load(fp) — десериализует Python объект из файла
- loads(s) — десериализует Python объект из строки

Дополнительные аргументы в методы можете передавать какие хотите :)

Сериализация/десериализация :

- класса
- объекта с простыми полями
- объекта со сложными полями и функциями
- функции

Консольная утилита должна работать следующим образом:

Конвертация сериализованных объектов из одного поддерживаемого формата в другой. Путь к файлу (файлам) указывается относительным или абсолютным путем, отдельным параметром передается новый формат. При указании исходного формата конвертирование не должно выполняться.

В случае передачи параметром файла конфигурации, вся информация должна браться оттуда и все остальные параметры проигнорированы.

Требования к программе

Разрешается использовать только стандартную библиотеку Python.
Пишем на версии Python 3.8+

Режимы работы:

- как библиотека, для переиспользования основной логики или вспомогательных функций
- как консольная утилита

Задание конфигурации:

- Возможность передачи всех конфигурационных параметров по отдельности через аргументы командной строки
- Опциональный конфигурационный файл для конкретного запуска через аргументы командной строки

Внутреннее устройство:

- Работа с аргументами командной строки с помощью модуля *argparse*
- Структура программы должна быть разбита на модули
- Реализовать программу так, чтобы отдельный полезный функционал можно было бы переиспользовать, пользуясь программой как библиотекой

Защита от ошибок и тестирование:

- Основная функциональность должна быть покрыта юнит-тестами. Тесты запускают программу в различных режимах работы и проверяют результаты. Примеры фреймворков: pytest, nose, unittest и другие какие хотите.
- Coverage должен быть 90+ %
- Если ваша программа падает во время тестов или при сдаче преподавателю, то лабораторная не засчитывается.

Установка:

- С помощью setup.py.

Критерии приема и сдачи лабораторной (Если хотя бы 1 пункт не выполняется, то лабораторная не засчитывается):

- Хорошие знания в теории по темам лабораторной
- Понимание того, что написано у вас в коде
- Покрытие тестами

Теория и практика к защите лабораторной

1. Темы рассмотренные на лекциях.

2. Практические задания ниже для самостоятельного решения и разбора: эти задания могут спросить на защите и по ним, и связанным темам, также могут быть вопросы.

Задания для самостоятельной подготовки

1.

Статистики по тексту. На вход поступают текстовые данные. Необходимо посчитать и вывести:

- сколько раз повторяется каждое слово в указанном тексте
- среднее количество слов в предложении
- медианное количество слов в предложении
- top-K самых часто повторяющихся буквенных N-грам (K и N имеют значения по-умолчанию 10 и 4, но должна быть возможность задавать их с клавиатуры)

При решении использовать контейнер dict() или его аналоги и встроенные операции над строками. Предусмотреть обработку знаков препинания.

2.

Генератор чисел Фибоначчи. Написать генератор возвращающий последовательно числа Фибоначчи начиная с первого.

3.

Изучить модуль re и написать регулярные выражения для:

- валидации email-адреса
- валидации записи числа с плавающей строчкой
- получения отдельных частей URL (схема, хост, порт, путь, параметры) с помощью механизма именованных групп

4.

Класс “n-мерный вектор”. У этого класса должны быть определены все естественные для вектора операции – сложение, вычитание, умножение на константу и скалярное произведение, сравнение на равенство(должно вызываться сравнение только 1 раз). Кроме этого должны быть операции вычисления длины, получение элемента по индексу, а также строковое представление.

5.

Класс логгер с возможностью наследования. Класс должен логировать то, какие методы и с какими аргументами у него вызывались и какой был результат этого вызова. Функция str() от этого класса должна отдавать лог вызовов. Должна быть возможность наследоваться от такого класса, чтобы добавить логирование вызовов у любого класса. При форматировании строк использовать метод format.

6.

Рекурсивный defaultdict. Реализовать свой класс-аналог defaultdict, который позволяет рекурсивно читать и записывать значения в виде d[“a”][“b”] = 1, а при вызове str(d) выводит данные как словарь в текстовом представлении.

7.

Метакласс берущий поля класса из файла. Реализовать метакласс, который позволяет при создании класса добавлять к нему произвольные атрибуты (классу, не экземпляру класса), которые загружаются из файла.
В файле должны быть имена атрибутов и их значения. Нужно уметь передавать путь к файлу как изменяемый параметр.

8.

Декоратор @cached, который сохраняет значение функции при каждом вызове. Если функция вызвана повторно с теми же аргументами, то возвращается сохраненное значение, а функция не вычисляется. Учесть как позиционные, так и именованные аргументы.

9.

Свой range. Реализовать полностью свой range с аналогичным встроенному интерфейсом.

10.

Последовательность с фильтрацией. Реализовать класс, соответствующий некоторой последовательности объектов и имеющий следующие методы:

- Создать объект на основе произвольного iterable объекта.
- Итерирование (__iter__) по элементам (неистощимое – можно несколько раз использовать объект в качестве iterable для for).

- Отфильтровать последовательность с помощью некоторой функции и вернуть новую сокращенную последовательность, в которой присутствуют только элементы, для которых эта функция вернула True.

11.

Синглтон. Реализовать шаблон проектирования Singleton, который можно применять на произвольный класс. Разработать самостоятельно, как этот инструмент будет применяться к целевому классу (например, модифицировать исходный класс или изменять способ вызова конструктора).

12.

Метакласс model creator. Реализуйте метакласс ModelCreator, позволяющий объявлять поля класса в следующем виде:

```
class Student (metaclass=ModelCreator):
    name = StringField()
```

Здесь StringField — некоторый объект, который обозначает, что это поле является текстовым. После такого вызова должен быть создан класс, конструктор которого принимает именованный аргумент name и сохраняет его в соответствующий атрибут (с возможной проверкой и приведением типа). Таким образом должна быть возможность писать так:

```
s = Student(name ='abc')
print(s.name)
```

Постарайтесь добиться того, чтобы создание класса было как можно более гибким: чтобы допускалось наследование и т.п. Обязательно должна быть проверка типов (например, в текстовое поле нельзя записать число).

13.

Сортировки. Реализовать следующие сортировки:

- быстрая
- слиянием
- поразрядная

Знать и понимать алгоритмическую сложность реализованных алгоритмов

14.

Хранилище уникальных элементов. При запуске программа работает в

интерактивном режиме и поддерживает команды:

- add <key> [<key> ...] - добавить один или более элементов в хранилище (если уже содержится, то не добавлять).
- remove <key> - удалить элемент из хранилища
- find <key> [<key> ...] - проверить наличие одного или более элементов в хранилище, вывести найденные
- list - вывести все элементы в хранилище
- grep <regexp> - поиск значения по регулярному выражению
- save и load - сохранить хранилище в файл и загрузить хранилище из файла

При решении использовать контейнер set().