Национальный Исследовательский Университет ИТМО
Факультет Программной Инженерии и Компьютерной Техники

Вариант №(13%35)=13
Лабораторная работа №4
по дисциплине
'Информатика'

Выполнил:
Студент группы P3113
***Крутько Никита* : 242570**
Преподаватель:
Малышева Татьяна Алексеевна

Санкт-Петербург 2019 г.

Содержание

1 Задание **2**

2 Исходный файл JSON **2**

3 Исходный код Python **3**
3.1 json_lib 3
3.2 yaml_lib 8

4 Результирующий файл YAML **10**

5 Вывод **11**

1. Задание

1. Исходя из структуры расписания конкретного дня, сформировать файл с расписанием в формате, указанном в задании в качестве исходного.
2. Написать программу на языке Python 3.x, которая бы осуществляла парсинг и конвертацию исходного файла в новый.
3. Нельзя использовать готовые библиотеки, кроме re (регулярные выражения в Python) и библиотеки для загрузки XML-файлов.
4. Необязательное задание для получения оценки «4» и «5» (позволяет набрать от 75 до 89 процентов от максимального числа баллов БаРС за данную лабораторную).
   (a) Найти готовые библиотеки, осуществляющие аналогичный парсинг и конвертацию файлов.
   (b) Сравнить полученные результаты и объяснить их сходство/различие.
5. Необязательное задание для получения оценки «5» (позволяет набратьот 90 до 100 процентов от максимального числа баллов БаРС за данную лабораторную).
   (a) Используя свою программу и найденные готовые библиотеки, сравнить десятикратное время выполнения парсинга + конвертации в цикле.
   (b) Проанализировать полученные результаты и объяснить их сходство/различие.

Таблица 1: Вариант задания

№ варианта	Исходный формат	Результирующий формат	День недели
13	JSON	YAML	Среда

2. Исходный файл JSON

```
{
    "day_id": 3,
    "group_id": "p3113",
    "schedule": [
        {
```

```
            "id": 25,
            "type_id": 0,
            "is_even": null,
            "time_id": 1,
            "campus_id": 2,
            "auditory_id": 2219,
            "personality_id": 153941
        },
        {
            "id": 32,
            "type_id": 1,
            "is_even": null,
            "time_id": 2w,
            "campus_id": 2,
            "auditory_id": null,
            "personality_id": null
        },
        {
            "id": 25,
            "type_id": 1,
            "is_even": null,
            "time_id": 3,
            "campus_id": 2,
            "auditory_id": 1122,
            "personality_id": 178308
        }
    ]
}
```

3. Исходный код Python

3.1. json_lib

```
from enum import Enum, auto
from . import parser


class JsonObject:
    """
    Class which represents JsonObject with form of object, not normalized
    """
    def __init__(self, data):
        self._data = data

    def from_string(self, string):
        """Converts string to JsonObject
        Gets: string
        Returns: JsonObject
        """
        return parser.Parser(string).parse()

    def to_string(self, to_formated=False, tabs=1):
        """Converts JsonObject to string
        Gets: self
        Returns: string
        """
        return self._data.to_string(to_formated, tabs)

    def to_normal(self):
        """Converts JsonObject to normal python object
```

```
        Gets: JsonObject
        Returns: normal value
        """
        return self._data.normalized()

    def from_normal(normal):
        """Converts normal python object to jsonobject
        Gets: normal value
        Returns: JsonObject
        """
        return JsonObject(Data.from_normal(normal))


class Data:
    def __init__(self, data_type, data):
        self._data, self._data_type, self._name = data, data_type, None

    def with_name(self, name):
        self._name = name
        return self

    def get_name(self):
        return self._name

    def normalized(self):
        if self._data_type == DataType.NULL:
            return None
        if self._data_type == DataType.NUMBER:
            return (int(self._data)
                    if int(self._data) == float(self._data)
                    else float(self._data))
        elif self._data_type == DataType.STRING:
            return str(self._data)
        elif self._data_type == DataType.BOOLEAN:
            return bool(self._data)
        elif self._data_type == DataType.ARRAY:
            return [val.normalized() for val in self._data]
        elif self._data_type == DataType.OBJECT:
            return dict((key, val.normalized())
                        for (key, val) in self._data.items())

    def from_normal(obj):
        data_type = DataType.to_type(obj)
        data = obj
        if data_type == DataType.ARRAY:
            data = [Data.from_normal(val) for val in obj]
        elif data_type == DataType.OBJECT:
            data = dict((key, Data.from_normal(val))
                        for (key, val) in obj.items())
        return Data(data_type, data)

    def to_string(self, to_formated=False, tabs=1):
        def pre(tabs):
            return '' if not to_formated else ("  " * tabs)

        def format_obj(begin, end):
            return "{}{}{}".format(
                begin + ('' if not to_formated
                         else ('' if len(self._data) == 0
                               else ("\n" + pre(tabs)))),
                (',' + (' ' if not to_formated else ("\n" + pre(tabs)))).join(
```

```
                    [val.to_string(to_formated, tabs + 1)
                     for val in self._data]
                    if self._data_type == DataType.ARRAY
                    else ["'{}': {}".format(key, val.to_string(to_formated,
                                                               tabs + 1))
                          for (key, val) in self._data.items()]),
                ('' if not to_formated
                 else ('' if len(self._data) == 0
                       else ("\n" + pre(tabs - 1)))) + end)
        if self._data_type == DataType.ARRAY:
            return format_obj('[', ']')
        elif self._data_type == DataType.OBJECT:
            return format_obj('{', '}')
        elif self._data_type == DataType.NULL:
            return "null"
        elif self._data_type == DataType.STRING:
            return '"{}"'.format(self._data)
        elif self._data_type == DataType.BOOLEAN:
            return "true" if self._data else "false"
        else:
            return str(self._data)


class DataType(Enum):
    NULL = None
    NUMBER = auto()
    STRING = auto()
    BOOLEAN = auto()
    ARRAY = auto()
    OBJECT = auto()

    def to_type(value):
        data_type = DataType.NULL
        if isinstance(value, bool):
            data_type = DataType.BOOLEAN
        elif isinstance(value, int) or isinstance(value, float):
            data_type = DataType.NUMBER
        elif isinstance(value, str):
            data_type = DataType.STRING
        elif isinstance(value, type([])):
            data_type = DataType.ARRAY
        elif isinstance(value, type({})):
            data_type = DataType.OBJECT
        return data_type


if __name__ == "__main__":
    print("<<<<BEGIN TESTING 'json_lib' MODULE>>>>")
    a = {"int": 666,
         "float": 666.777,
         "string": "String",
         "array": ["String", 0o666, {"1": True, "2": [0x666, 0o777]}],
         "empty seq": [],
         "empty map": {}}
    print("<<INPUT>>")
    print(a)
    print("\n<<PRINTING JsonObject FORMATTED>>")
    print(JsonObject.from_normal(a).to_string(to_formated=True))
    print("\n<<PRINTING JsonObject PLAIN>>")
    print(JsonObject.from_normal(a).to_string())
    print("\n<<<<END TESTING 'json_lib' MODULE>>>>")
```

Listing 1: __init__.py

```
import json_lib as jl
from collections import deque
import re


class Parser:
    def __init__(self, string):
        self._string = string
        self._stack = deque()
        self._result = None

    def str_to_bool(value):
        return True if value == "true" else False

    def parse(self):
        while not self._result:
            self._parse_data()
            self._string = self._string[len(self._match.group(0)):]
        if self._string.strip().rstrip() != "":
            raise ValueError
        return jl.JsonObject(self._result)

    def _parse_data(self):
        """Parses string and founds first Data
        Gets: self
        Returns: None
        Throws: Exception, if can't parse data
        """
        regex_begin = r"^\s*(?P<val>"
        regex_end = r")\s*"
        if len(self._stack) != 0:
            if self._stack[-1]._data_type == jl.DataType.OBJECT:
                self._head_type = jl.DataType.OBJECT
                regex_begin = "^{}{}{}{}{}".format(
                    r"\s*((",
                    r",\s*(" if len(self._stack[-1]._data) != 0 else "(",
                    r"?P<name>",
                    DataRegex.STRING,
                    r")\s*:\s*(?P<val>"
                )
                regex_end = r"))|(\}))\s*"
            elif self._stack[-1]._data_type == jl.DataType.ARRAY:
                self._head_type = jl.DataType.ARRAY
                regex_begin = "^{}{}{}".format(
                    r"\s*(",
                    r",\s*(" if len(self._stack[-1]._data) != 0 else "(",
                    r"?P<val>"
                )
                regex_end = r")|([\]]))\s*"
            else:
                self._head_type = None
        else:
            self._head_type = None

        any_matches = False
        for (data_type, data_regex) in DataRegex.get_all().items():
            if self._match_string(regex_begin, data_regex, regex_end):
```

```
                self._read_from_match(data_type)
                any_matches = True
                break
        if not any_matches:
            raise ValueError

    def _match_string(self, regex_begin, regex, regex_end):
        self._match = re.compile(
            "{}{}{}".format(regex_begin,
                            regex,
                            regex_end),
            re.DOTALL).search(self._string)
        return True if self._match else False

    def _read_from_match(self, data_type):
        def string_to_jsonobj(data_type):
            if data_type == jl.DataType.ARRAY:
                return jl.Data.from_normal([])
            elif data_type == jl.DataType.OBJECT:
                return jl.Data.from_normal({})
            elif data_type == jl.DataType.STRING:
                return jl.Data.from_normal(self._match.groupdict()['val'][1:-1])
            elif data_type == jl.DataType.NUMBER:
                return jl.Data.from_normal(
                    float(self._match.groupdict()['val']))
            elif data_type == jl.DataType.BOOLEAN:
                return jl.Data.from_normal(
                    Parser.str_to_bool(self._match.groupdict()['val']))
            elif data_type == jl.DataType.NULL:
                return jl.Data.from_normal(None)
        if self._match.groups()[-1] and (
                self._head_type == jl.DataType.OBJECT or
                self._head_type == jl.DataType.ARRAY):
            if len(self._stack) >= 2:
                if self._stack[-2]._data_type == jl.DataType.OBJECT:
                    name = self._stack[-1]._name[1:-1]
                    self._stack[-1]._data[name] = self._stack.pop()
                else:
                    self._stack[-2]._data.append(self._stack.pop())
            else:
                self._result = self._stack.pop()
        elif self._head_type == jl.DataType.OBJECT and (
                data_type in [jl.DataType.ARRAY, jl.DataType.OBJECT]):
            self._stack.append(string_to_jsonobj(data_type).with_name(
                self._match.groupdict()['name']))

        elif self._head_type == jl.DataType.OBJECT:
            self._stack[-1]._data[self._match.groupdict()['name'][1:-1]] = (
                string_to_jsonobj(data_type))

        elif self._head_type == jl.DataType.ARRAY and (
                data_type in [jl.DataType.ARRAY, jl.DataType.OBJECT]):
            self._stack.append(string_to_jsonobj(data_type))

        elif self._head_type == jl.DataType.ARRAY:
            self._stack[-1]._data.append(string_to_jsonobj(data_type))

        elif data_type == jl.DataType.ARRAY or data_type == jl.DataType.OBJECT:
            self._stack.append(string_to_jsonobj(data_type))

        else:
```

```
            self._result = string_to_jsonobj(data_type)


class DataRegex:
    NULL = r"null"
    BOOLEAN = r"(true)|(false)"
    NUMBER = r"-?\d+(.\d+([eE][-+]?\d+)?)?"
    STRING = (
        r"[\'\"](((\\([\"\\\/bfnrt])|(u[0-9A-Fa-f]{4}))|[^\'\"\\])*)([^\\][\'\"])")
    ARRAY = r"\["
    OBJECT = r"\{"

    def get_all():
        return {
            jl.DataType.ARRAY: DataRegex.ARRAY,
            jl.DataType.OBJECT: DataRegex.OBJECT,
            jl.DataType.STRING: DataRegex.STRING,
            jl.DataType.NUMBER: DataRegex.NUMBER,
            jl.DataType.BOOLEAN: DataRegex.BOOLEAN,
            jl.DataType.NULL: DataRegex.NULL
        }
```

Listing 2: parser.py

3.2. yaml_lib

```
from enum import Enum, auto


class YamlObject:
    """
    Class which represents YamlObject with form of object, not normalized
    """
    def __init__(self, data):
        self._data = data

    # def from_string(self, string):
    #     """Converts string to YamlObject
    #     Gets: string
    #     Returns: YamlObject
    #     """
    #     return yaml_object

    def to_string(self, tabs=0):
        """Converts YamlObject to string
        Gets: self
        Returns: string
        """
        return self._data.to_string(tabs)

    def to_normal(self):
        """Converts YamlObject to normal python object
        Gets: YamlObject
        Returns: normal value
        """
        return self._data.normalized()

    def from_normal(normal):
        """Converts normal python object to YamlObject
        Gets: normal value
```

```
        Returns: YamlObject
        """
        return YamlObject(Data.from_normal(normal))


class Data:
    def __init__(self, data_type, data):
        self._data, self._data_type = data, data_type

    def get_name(self):
        return self._name

    def normalized(self):
        if self._data_type == DataType.NULL:
            return None
        elif self._data_type == DataType.BOOL:
            return bool(self._data)
        elif self._data_type == DataType.INT_8:
            return oct(self._data)
        elif self._data_type == DataType.INT_10:
            return int(self._data)
        elif self._data_type == DataType.INT_16:
            return int(self._data, 16)
        elif self._data_type == DataType.FLOAT:
            return float(self._data)
        elif self._data_type == DataType.STR:
            return str(self._data)
        elif self._data_type == DataType.SEQ:
            return [val.normalized() for val in self._data]
        elif self._data_type == DataType.MAP:
            return dict((key, val.normalized())
                        for (key, val) in self._data.items())

    def from_normal(obj):
        data_type = DataType.to_type(obj)
        data = obj
        if data_type == DataType.SEQ:
            data = [Data.from_normal(val) for val in obj]
        elif data_type == DataType.MAP:
            data = dict((key, Data.from_normal(val))
                        for (key, val) in obj.items())
        return Data(data_type, data)

    def to_string(self, tabs=0):
        def pre(tabs):
            return "  " * tabs

        if ((self._data_type == DataType.SEQ or
             self._data_type == DataType.MAP) and len(self._data) == 0):
            return "[]" if self._data_type == DataType.SEQ else "{}"
        elif self._data_type == DataType.SEQ:
            return '\n' + '\n'.join(["{}- {}".format(pre(tabs),
                                                     val.to_string(tabs + 1))
                                     for val in self._data])
        elif self._data_type == DataType.MAP:
                return '\n{}'.format(pre(tabs)).join(
                    ["{}: {}".format(key, val.to_string(tabs + 1))
                     for (key, val) in self._data.items()])
        elif self._data_type == DataType.NULL:
            return "Null"
        elif self._data_type == DataType.STR:
```

```
            if len(self._data.split('\n')) > 1:
                return "|\n" + '\n'.join(pre(tabs + 1) + val
                                         for val in self._data.split('\n'))
            else:
                return self._data
        elif self._data_type == DataType.INT_8:
            return str(oct(self._data))
        elif self._data_type == DataType.INT_16:
            return str(hex(self._data))
        else:
            return str(self._data)


class DataType(Enum):
    NULL = auto()
    BOOL = auto()
    INT_10 = auto()
    INT_8 = auto()
    INT_16 = auto()
    FLOAT = auto()
    STR = auto()
    SEQ = auto()
    MAP = auto()

    def is_hex(s):
        try:
            int(s, 16)
            return True
        except ValueError:
            return False

    def to_type(value):
        data_type = DataType.NULL
        if isinstance(value, bool):
            data_type = DataType.BOOL
        elif isinstance(value, int):
            data_type = DataType.INT_10
        elif isinstance(value, float):
            data_type = DataType.FLOAT
        elif isinstance(value, str):
            data_type = DataType.STR
        elif isinstance(value, type([])):
            data_type = DataType.SEQ
        elif isinstance(value, type({})):
            data_type = DataType.MAP
        return data_type


if __name__ == "__main__":
    print("<<<<BEGIN TESTING 'yaml_lib' MODULE>>>>")
    a = {"int": 666,
         "float": 666.777,
         "string": "String",
         "array": ["String", 0o666, {"1": True, "2": [0x666, 0o777]}],
         "empty seq": [],
         "empty map": {}}
    print("<<INPUT>>")
    print(a)
    yaml = YamlObject.from_normal(a)
    print("\n<<PRINTING YamlObject TO_STRING>>")
    print(yaml.to_string())
```

```
    print("\n<<PRINTING YamlObject NORMALIZED>>")
    print(yaml.to_normal())
    print("\n<<<<END TESTING 'json_lib' MODULE>>>>")
```

Listing 3: __init__.py

4. Результирующий файл YAML

```yaml
day_id: 3
group_id: p3113
schedule:
  - id: 25
    type_id: 0
    is_even: Null
    time_id: 1
    campus_id: 2
    auditory_id: 2219
    personality_id: 153941
  - id: 32
    type_id: 1
    is_even: Null
    time_id: 2
    campus_id: 2
    auditory_id: Null
    personality_id: Null
  - id: 25
    type_id: 1
    is_even: Null
    time_id: 3
    campus_id: 2
    auditory_id: 1122
    personality_id: 178308
```

5. Вывод

Много кода на Python и довольно удачный парсер JSON через регулярки, в принципе должен нормально сжевать и переварить любой JSON файл, в случае, если не может распарсить его (т.е. если файл не валидный), то кидает эксепшн ValueError. Также сделал небольшие настройки для YAML и JSON