РОБОТА З СЕРІАЛІЗАЦІЙНИМ ПРЕДСТАВЛЕННЯМ JSON

Питання 10.4.

Запис (dumping) та завантаження (loading) даних у форматі JSON

- JSON (JavaScript Object Notation) – легковаговий формат обміну даними.
  - Простий для читання/запису людьми та розбору/генерації комп'ютером.
  - Базується на підмножині мови програмування JavaScript, Standard ECMA-262 3rd Edition - грудень 1999.
  - JSON – текстовий формат, незалежний від мови програмування, проте він використовує угоди, знайомі для програмістів С-подібними мовами.
- Формат використовується широким спектром мов та фреймворків.
  - Такі бази даних, як CouchDB, представляють дані у вигляді JSON-об'єктів, спрощуючи передачу даних між додатками.
  - JSON-документи дуже схожі на списки та словники в Python.
- Модуль json працює із вбудованими типами Python.
  - Без додаткових змін підтримки власних типів розробника немає.
  - Існує відображення на типи JavaScript, які використовує JSON.
  - Інші типи не підтримуються і повинні зводитись до одного з представлених вище за допомогою extension-функцій, які можна включити в функції dump() і load().

dict	object
list, tuple	array
str	string
int, float	number
True	true
False	false
None	null

Оголосимо класи для підтримки персистентності. Розглянемо простий мікроблог та пости в ньому

```python
import datetime

class Post:
    def __init__( self, date, title, rst_text, tags ):
        self.date= date
        self.title= title
        self.rst_text= rst_text
        self.tags= tags

    def as_dict( self ):
        return dict( date= str(self.date), title= self.title,
                     underline= "-"*len(self.title),
                     rst_text= self.rst_text,
                     tag_text= " ".join(self.tags),
        )
```

- Для підтримки простої підстановки в шаблони метод as_dict() повертає словник значень, які були зведені у рядковий формат.
  - Додатково введено кілька значень, щоб допомогти вивести RST.
  - Атрибут tag_text is – сплюснута текстова версія кортежу тегів.
  - Атрибут underline утворює underline рядок з довжиною, яка відповідає довжині заголовку; це допомагає при RST-форматуванні.
- Також створимо блог як колекцію постів.
  - Маємо три варіанти проєктування: обгорнути, розширити або написати новий клас.
  - Розширення ітерованих об’єктів ускладнюється тим, що вони мають вбудовані алгоритми серіалізації, робота яких може порушитись через додавання нових можливостей.
  - Для постійного зберігання розширена версія списку не підійде!

Обгортання чи створення власного класу?

```python
from collections import defaultdict

class Blog:
    def __init__( self, title, posts=None ):
        self.title= title
        self.entries= posts if posts is not None else []

    def append( self, post ):
        self.entries.append(post)

    def by_tag(self):
        tag_index= defaultdict(list)
        for post in self.entries:
            for tag in post.tags:
                tag_index[tag].append( post.as_dict() )
        return tag_index

    def as_dict( self ):
        return dict(title= self.title, underline= "="*len(self.title),
                    entries= [p.as_dict() for p in self.entries],)
```

- Також включено атрибут title – заголовок мікроблогу.
  - Якщо posts=None, використовуємо порожній список.
  - Інакше беремо надане значення для постів.
- Додатково визначено метод, який індексує пости за тегами.
  - У результуючому defaultdict кожен ключ є текстом тегу, а кожне значення – списком постів, які володіють цим тегом.
- Для спрощення використання шаблонних рядків включено метод as_dict(), який збирає весь блог у простий словник рядків та словників.
  - Ідея: формувати лише вбудовані типи, які мають просте рядкове представлення.

Процес рендерингу шаблону

```
travel = Blog( "Travel" )
travel.append(Post( date=datetime.datetime(2013,11,14,17,25), title="Hard Aground",
                    rst_text="""Some embarrassing revelation. Including ☹ and □""",
                    tags=("#RedRanger", "#Whitby42", "#ICW"),))
travel.append(Post( date=datetime.datetime(2013,11,18,15,30), title="Anchor Follies",
                    rst_text="""Some witty epigram. Including < & > characters.""",
                    tags=("#RedRanger", "#Whitby42", "#Mistakes"),))
```

- Потрібно серіалізувати Blog і Post як Python-код.
  - Це не поганий спосыб представлення блогу, у деяких випадках Python-код доречний для представлення об'єкта.
- Для повноти викладу представимо інструмент для перетворення RST-виводу в HTML-файл.
  - Це може зробити rst2html.py з docutils.
- Існує багато додаткових template tools, які можуть виконати більш складну підстановку, включаючи цикли та галуження всередині самого шаблону.
  - Список альтернатив: https://wiki.python.org/moin/Templating.
  - Розглянемо приклад з Jinja2 template tool.

Процес рендерингу шаблону

```
from jinja2 import Template
blog_template= Template( """
{{title}}
{{underline}}
{% for e in entries %}
{{e.title}}
{{e.underline}}
{{e.rst_text}}
:date: {{e.date}}
:tags: {{e.tag_text}}
{% endfor %}
Tag Index
=========
{% for t in tags %}
* {{t}}
{% for post in tags[t] %}
- `{{post.title}}`_
{% endfor %}
{% endfor %}
""")
print( blog_template.render( tags=travel.by_tag(), **travel.as_dict()) )
```

- Метод render() викликається з **travel.as_dict(), щоб перевірити такі атрибути, як title і underline, на відповідність іменованим аргументам.
- Оператори {%for%} та {%endfor%} показують, як Jinja може ітерувати по послідовності постів у блозі.
  - У тілі циклу змінна ***e*** буде словником, створеним з кожного посту.
  - Було обрано ключі зі словників постів: {{e.title}}, {{e.rst_text}} та ін.
- Також ітеруємо по колекції тегів для блогу.
  - Це словник з ключами для тегу та значенням посту.
  - Цикл пройде по кожному ключу, присвоєному в ***t***.
  - Тіло циклу виконує прохід по постах (значеннях у словнику), ***tags[t]***.
- Оператор {{post.title}} – розмітка RST, що генерує лінк на секцію, що має той же заголовок у документі.
  - Заголовки блогів використовувались як секції та лінки в індексі. Заголовки повинні бути унікальними, інакше отримаємо помилки рендерингу RST.
  - Оскільки цей шаблон проходить по всьому блогу, всі пости будуть відформатовані за один раз.
  - Вбудований у Python шаблон string.Template не може ітерувати, що ускладнює рендеринг постів блогу.

Перетворення об’єктів мікроблогу в простіші об’єкти Python: списки та словники

- import json
  print(json.dumps(travel.as_dict(), indent=4))
- Вивід у JSON-форматі


```
{
    "underline": "======",
    "entries": [
        {
            "tag_text": "#RedRanger #Whitby42 #ICW",
            "underline": "------------",
            "title": "Hard Aground",
            "date": "2013-11-14 17:25:00",
            "rst_text": "Some embarrassing revelation. Including \u2639 and \u2395"
        },
        {
            "tag_text": "#RedRanger #Whitby42 #Mistakes",
            "underline": "--------------",
            "title": "Anchor Follies",
            "date": "2013-11-18 15:30:00",
            "rst_text": "Some witty epigram. Including < & > characters."
        }
    ],
    "title": "Travel"
}
```

Деякі недоліки JSON-представлення

- Потрібно переписати наші Python-об’єкти у словники.
  - Використовуючи json.load(), не отримаємо об’єкти Blog або Post; будуть словники та списки.
  - Існують деякі значення в __dict__ об’єкта, які краще не зберігати персистентно, зокрема underlined-текст у Post-об’єкта.
  - Для кодування наших об’єктів у JSON необхідно надати функцію, яка спростить об’єкти до примітивних типів Python. Вона називається ***функцією за умовчанням (default function)*** і постачає кодування за умовчанням (default encoding) для об’єкта невідомого класу.
  - Для декодування об’єктів з JSON необхідно мати функцію, яка перетворить словник примітивних типів Python назад в об’єкт потрібного класу. Вона називається ***функцією отримання об’єкта (object hook function).***

Підтримка JSON у наших класах

- Документація модуля json передбачає, що може виникнути потреба у використанні ***підказок (class hinting)***.
  - Пропозиція: кодувати екземпляр власного класу у вигляді словника наступним чином:
  - {"__jsonclass__": ["class name", [param1,...]] }
  - Запропоноване значення, пов'язане з ключем "__jsonclass__", є списком з двох елементів: іменем класу та списком аргументів, що потрібні для створення екземпляру класу.
  - Специфікація JSON-RPC дозволяє більше можливостей, проте вони не є релевантними для Python.
- Для декодування об'єкта із JSON-словника можна шукати ключ "__jsonclass__" як підказку, що потрібно збирати (build) один з наших класів, а не вбудований Python-об'єкт.
  - Назва класу може відображатись на об'єкт класу, а послідовність аргументів може використовуватись для конструювання екземпляру.
  - Інші потужні JSON-кодувальники (зокрема з вебфреймворку Django), забезпечують більш складне кодування.

Кастомізація JSON-кодування

- Включимо ключ __class__, який називатиме цільовий клас.
  - Ключ __args__ надаватиме послідовність значень позиційних аргументів.
  - Ключ __kw__ постачатиме словник значень іменованих аргументів.
- Передаємо функцію blog_encode() у функцію json.dumps().
  - text= json.dumps(travel, indent=4, default=blog_encode)
  - Вона використовується кодувальником JSON, щоб визначити кодування об'єкта.

```
def blog_encode( object ):
    if isinstance(object, datetime.datetime):
        return dict( __class__= "datetime.datetime", __args__= [],
                     __kw__= dict( year= object.year, month= object.month,
                                   day= object.day, hour= object.hour,
                                   minute= object.minute, second= object.second,))
    elif isinstance(object, Post):
        return dict(__class__= "Post", __args__= [],
                     __kw__= dict(date= object.date, title= object.title,
                                   rst_text= object.rst_text, tags= object.tags,))
    elif isinstance(object, Blog):
        return dict(__class__= "Blog", __args__= [object.title, object.entries,],
                     __kw__= {})
    else:
        return json.JSONEncoder.default(o)
```

```
{
    "__args__": [
        "Travel",
        [
            {
                "__args__": [],
                "__kw__": {
                    "tags": [
                        "#RedRanger",
                        "#Whitby42",
                        "#ICW"
                    ],
                    "rst_text": "Some embarrassing revelation.
Including \u2639 and \u2693",
                    "date": {
                        "__args__": [],
                        "__kw__": {
                            "minute": 25,
                            "hour": 17,
                            "day": 14,
                            "month": 11,
                            "year": 2013,
                            "second": 0
                        },
                        "__class__": "datetime.datetime"
                    },
                    "title": "Hard Aground"
                },
                "__class__": "Post"
            },
.
.
.
    "__kw__": {},
    "__class__": "Blog"
}
```

Кастомізація декодування JSON

- Для декодування JSON-об'єкта потрібно розбирати структуру JSON-документу.
  - Об'єкти нашого класу кодувались у вигляді словників.
  - Кожен dict, отриманий від декодера JSON ***може*** бути одним з наших класів.
- «object hook» декодера JSON – це функція, що викликається для кожного словника, щоб перевірити, чи представляє він наш об'єкт.
  - Якщо dict не розпізнається hook-функцією, тоді це просто словник, він буде повертатись без змін.

```
def blog_decode( some_dict ):
    if set(some_dict.keys()) == set( ["__class__", "__args__", "__kw__"]
        class_= eval(some_dict['__class__'])
        return class_( *some_dict['__args__'], **some_dict['__kw__'] )
    else:
        return some_dict
```

- Кожного разу, коли функція викликається, перевіряються ключі, які визначають кодування для наших об'єктів.
  - Якщо присутні 3 ключі, дана функція викликається подібним чином:
    blog_data= json.loads(text, object_hook= blog_decode)

Проблеми з безпекою та eval()

- потенційна безпекова проблема, якщо шкідливий код вписано в саме JSON-представлення.
  - Це загальна проблема з інтернет-документами, а не eval().
- Слід усувати проблему, коли недостовірний документ було скомпрометовано атакою посередника.
  - Тоді JSON-документ змінюється при проходженні через веб-інтерфейс, який містить недостовірний сервер, що поводить себе як проксі. SSL – поширений метод для уникнення проблеми.
- За потреби можна замінити eval() на словник з відображенням «назва: клас».
  - Було: eval(some_dict['__class__'])
  - Стало: {"Post":Post, "Blog":Blog,"datetime.datetime":datetime.datetime: }[some_dict['__class__']]
  - Це запобігає проблемам проходження JSON-документа по не-SSL-шифрованому з'єднанню.
  - Також це веде потреби переналаштування відображення при проєктних змінах у додатку.

Рефакторинг функції encode()

- Потрібно реструктурувати кодуючу функцію, щоб сконцентруватись на правильному кодуванні для кожного визначеного класу.
  - Краще не накопичувати всі правила кодування в одній функції.
- Для бібліотечних класів, таких як datetime, буде потрібно розширити datetime.datetime для нашого додатку.
  - Матимемо 2 розширення класу, які створюватимуть JSON-кодовані описи класів.
  - Додамо властивість до класу Blog:

```
@property
def _json( self ):
    return dict( __class__= self.__class__.__name__,
                 __kw__= {},
                 __args__= [ self.title, self.entries ])
```

Властивість забезпечить аргументи для ініціалізації, які може використати декодуюча функція

- Спростимо кодувальник (encoder).

```
@property
def _json( self ):
    return dict(__class__= self.__class__.__name__,
                __kw__= dict(date= self.date,
                             title= self.title,
                             rst_text= self.rst_text,
                             tags= self.tags,),
                __args__= [])
```

```
def blog_encode_2( object ):
    if isinstance(object, datetime.datetime):
        return dict(__class__= "datetime.datetime", __args__= [],
                    __kw__= dict(year= object.year, month= object.month,
                                 day= object.day, hour= object.hour,
                                 minute= object.minute, second= object.second,
                    ))
    else:
        try:
            encoding= object._json()
        except AttributeError:
            encoding= json.JSONEncoder.default(o)
        return encoding
```

Стандартизація рядка з датою та загальний вивід

- Для кращої сумісності з іншими мовами бажано коректно записувати datetime-об’єкт у вигляді стандартного рядка, а потім парсити цей рядок:

```
if isinstance(object, datetime.datetime):
    fmt= "%Y-%m-%dT%H:%M:%S"
    return dict(__class__= "datetime.datetime.strptime",
                __args__= [ object.strftime(fmt), fmt ],
                __kw__= {})
```

- Закодований вивід вказує статичний метод datetime.datetime.strptime() та аргумент із закодованим datetime-об’єктом і форматом його декодування.
- Тепер маємо ISO-відформатовану дату замість індивідуальних полів.
- Відмовились від створення об’єкта через назву класу.
- Значення __class__ розширено до назви класу або статичного методу.

```
{
    "__args__": [],
    "__class__": "Post_J",
    "__kw__": {
        "title": "Anchor Follies",
        "tags": [
            "#RedRanger",
            "#Whitby42",
            "#Mistakes"
        ],
        "rst_text": "Some witty epigram.",
        "date": {
            "__args__": [
                "2013-11-18T15:30:00",
                "%Y-%m-%dT%H:%M:%S"
            ],
            "__class__": "datetime.datetime.strptime",
            "__kw__": {}
        }
    }
}
```

Запис JSON у файл

- Для запису JSON-файлів загалом виконуються подібні дії:

```
with open("temp.json", "w", encoding="UTF-8") as target:
        json.dump( travel3, target, separators=(',', ':'), default=blog_j2_encode )
```

- Для зчитування JSON-файлів:

```
with open("some_source.json", "r", encoding="UTF-8") as source:
        objects= json.load( source, object_hook= blog_decode)
```

Ідея: відокремити текстове JSON-представлення від будь-якого conversion to bytes on the resulting fle

- Існує кілька опцій форматування, доступних для JSON.
  - Використання відступів робить JSON-файл більш читабельним.
  - Альтернативний, більш компактний вивід (temp.json):


```
{"__class__":"Blog_J","__args__":["Travel",[{"__class__":"Post_J","__
args__":[],"__kw__":{"rst_text":"Some embarrassing revelati
on.","tags":["#RedRanger","#Whitby42","#ICW"],"title":"Hard
Aground","date":{"__class__":"datetime.datetime.strptime","__
args__":["2013-11-14T17:25:00","%Y-%m-%dT%H:%M:%S"],"__
kw__":{}}}},{"__class__":"Post_J","__args__":[],"__kw__":{"rst_
text":"Some witty epigram.","tags":["#RedRanger","#Whitby42","#Mistak
es"],"title":"Anchor Follies","date":{"__class__":"datetime.datetime.
strptime","__args__":["2013-11-18T15:30:00","%Y-%m-%dT%H:%M:%S"],"__
kw__":{}}}}]],"__kw__":{}}
```

ДЯКУЮ ЗА УВАГУ!

Наступна тема: Робота з серіалізаційним представленням csv