Серіалізація об’єктів

Серіалізація та десеріалізація

Для персистентного зберігання Python-об'єктів необхідно конвертувати його в байти та записати їх у файл.

- Цю операцію називають ***серіалізацією***, або ***маршалінгом*** (marshaling, deflating, encoding).

Кожну зі схем серіалізації також можна назвати фізичним форматом даних.

- Слід відрізняти його від логічного формату даних – простого впорядкування за допомогою пробілів, яке змінює послідовність байтів, проте не значення об'єкту.

За виключенням CSV, такі представлення націлені на збереження одного Python-об'єкта.

- Хоч один об'єкт може бути списком об'єктів, він буде конкретного розміру.
- Для обробки одного з об'єктів потрібно ***десеріалізувати*** весь список.

Серіалізаційні представлення

JavaScript Object Notation (JSON): поширене представлення. Стандартний модуль json постачає класи та функції, необхідні для завантаження та збереження (dump) даних тільки у JSON-форматі.

Yet Ain't Markup Language (YAML): розширення JSON, яке дещо спрощує серіалізований вивід. Спеціальний пакет PyYaml передбачає багато можливостей для підтримки персистентності Python.

pickle: модуль pickle має власне, специфічне для Python, представлення даних. Недоліком формату є слабкі можливості для обміну даними з програмами, написаними не мовою Python. Цей формат є базовим для модуля shelve та черг повідомлень (message queues).

Comma-Separated Values (CSV): поширений формат, проте може бути незручним для представлення складних Python-об'єктів. CSV дозволяє виконувати інкрементоване представлення колекцій Python-об'єктів, that cannot fit into memory.

XML: дуже поширений формат, тому його парсинг має велике значення. Модулів для цієї задачі багато, будемо розглядати ElementTree.

Python-об’єкти можуть жити, поки працює відповідний процес

Поки на них є посилання в просторі імен.

- Якщо бажаємо мати об’єкт, що існуватиме за межами процесу чи простору імен, необхідно зробити його ***персистентним***.
- Більшість ОС пропонують підтримку персистентності у формі файлової системи.

Складність: об’єкти можуть посилатись на інші об’єкти.

- Об’єкт посилається на свій клас, клас посилається на свій метаклас чи базовий клас.
- Об’єкт може бути контейнером та посилатись на інші об’єкти.
- Оскільки розташування в пам’яті не фіксовані, зв’язки між об’єктами порушуться після спроби збереження та відновлення байтів з пам’яті без переписування адрес у певного виду location-independent ключ.

Проблема схеми міграції (Schema Migration Problem)

Багато об’єктів у павутині посилань у значній мірі статичні.

- Наприклад, опис класів змінюється повільніше за значення змінних, а в ідеалі, не змінюється зовсім.
- Проте на рівні класу можуть бути присутні instance variables.
- Управління зміною структури (класу) даних називають Schema Migration Problem.

Python надає формальну відмінність між змінними об’єкта та іншими атрибутами класу.

- We define an object's instance variables to properly show the dynamic state of the object.
- Атрибути рівня класу використовують для інформації, що буде спільною для об’єктів цього класу.
- Якщо можна зберегти на постійній основі лише динамічний стан об’єкта—separated from the class and the web of references that are part of the class definition—that would be a workable solution to serialization and persistence.

Модуль pickle реалізує потужний алгоритм серіалізації/десеріалізації объектов Python

"Pickling" - процес перетворення об'єкта Python у потік байтів, а "unpickling" – зворотна операція.

```
>>> import pickle
>>> data = {
...     'a': [1, 2.0, 3, 4+6j],
...     'b': ("character string", b"byte string"),
...     'c': {None, True, False}
... }
>>>
>>> with open('data.pickle', 'wb') as f:
...     pickle.dump(data, f)
...
>>> with open('data.pickle', 'rb') as f:
...     data_new = pickle.load(f)
...
>>> print(data_new)
{'c': {False, True, None}, 'a': [1, 2.0, 3, (4+6j)], 'b': ('character string', b'byte string')}
```

Поширена термінологія для Python (dump і load)

Більшість класів у даній темі визначають наступні методи:

- dump(object, file): запис (dump) об'єкту *object* у заданий файл
- dumps(object): записує об'єкт, при цьому повертаючи рядкове представлення
- load(file): завантажує об'єкт із вказаного файлу, повертає сконструйований об'єкт
- loads(string): завантажує об'єкт з рядкового представлення, повертає сконструйований об'єкт.

Загалом файл, що використовувався для dump/load, може бути будь-яким файлоподібним об'єктом.

- Можна використовувати об'єкти io.StringIO, як і об'єкти urllib.request у якості джерел даних для завантаження.
- Аналогічно, dump накладає кілька вимог на джерело даних.

Файлова система та мережа

Оскільки файлова система ОС (і мережа) працюють з байтами, необхідно представляти значення полів об'єкта у якості серіалізованих потоків байтів.

- Часто застосовують двоетапне перетворення:
- Стан об'єкта представляється у вигляді рядка.
- Засоби Python конвертують рядки в байти.

У контексті файлових систем виділяють два класи пристроїв:

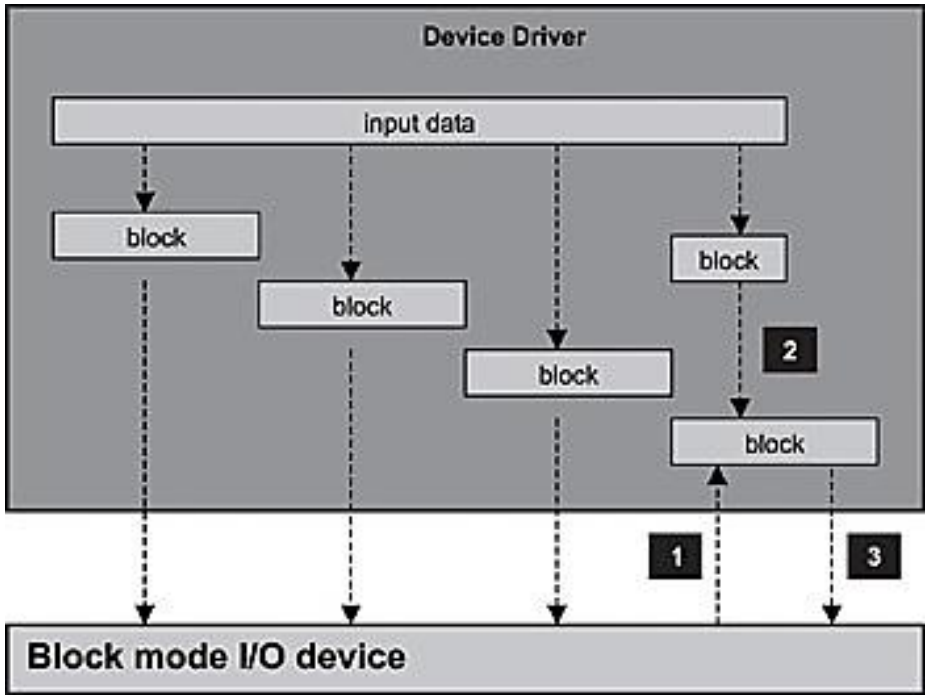
- Блочні пристрої (Block-mode devices) також називають seekable, оскільки ОС підтримує операції пошуку (seek operation), які можуть отримати доступ до будь-якого байту файлу.
- Character-mode devices не є seekable; це інтерфейси, в які байти послідовно передаються. Пошук може бути зворотним.

Відмінності між character та block mode можуть вплинути на представлення стану складного об'єкта або колекції об'єктів.

- Далі розглянемо підходи до серіалізації найпростішого набору даних – впорядкованого байтового потоку (stream of bytes);
- Ці формати make no use of seekable devices; вони збережуть байтовий потік або в символьний, або блочний файл.

Character-mode devices дозволяють передачу неструктурованих даних.

- *Зазвичай передача відбувається послідовно, по байту.*
- *Це прості пристрої, на зразок послідовного інтерфейсу чи клавіатури.*
- *Драйвер буферизує дані у випадках, коли швидкість передачі від системи до пристрою вища, ніж пристрій може handle.*



Block-mode devices передають дані блоками, наприклад, по 1Кб.

- *Розмір блоку визначається апаратним забезпеченням.*
- *Дані можуть бути або в деякій мірі структуровані, або передаватись по деякому протоколу передачі.*
- *Інакше ймовірна помилка.*
- *Тому інколи необхідно для драйверу блочного пристрою виконувати додаткову роботу для операції зчитування чи запису*

Оголосимо класи для підтримки персистентності

Розглянемо простий мікроблог та пости в ньому

```
1 import datetime
2
3 class Post:
4     def __init__( self, date, title, rst_text, tags ):
5         self.date= date
6         self.title= title
7         self.rst_text= rst_text
8         self.tags= tags
9
10    def as_dict( self ):
11        return dict( date= str(self.date), title= self.title,
12                     underline= "-"*len(self.title),
13                     rst_text= self.rst_text,
14                     tag_text= " ".join(self.tags),
15        )
```

Коментарі до коду

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 a flattened 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 проектування: 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?

```
1 from collections import defaultdict
2
3 class Blog:
4     def __init__( self, title, posts=None ):
5         self.title= title
6         self.entries= posts if posts is not None else []
7
8     def append( self, post ):
9         self.entries.append(post)
10
11    def by_tag(self):
12        tag_index= defaultdict(list)
13        for post in self.entries:
14            for tag in post.tags:
15                tag_index[tag].append( post.as_dict() )
16        return tag_index
17
18    def as_dict( self ):
19        return dict(title= self.title, underline= "="*len(self.title),
20                    entries= [p.as_dict() for p in self.entries],)
```

Крім обгортання списку, ще було включено атрибут – заголовок (title) мікроблогу.

- Ініціалізатор використовує поширену практику: не задавати значення за замовчуванням як змінюваний (mutable) об'єкт.
- Задаємо None: якщо пости 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.
- Кожне значення – список постів, які володіють цим тегом.

Для спрощення використання шаблонних рядків включено інший метод as_dict(), який збирає весь блог у простий словник рядків та словників.

- Ідея: to produce 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"),))
```

We've serialized the Blog and Post as the Python code.

- This isn't really all bad as a way to represent the blog.
- There are some use cases where Python code is a perfectly fne representation for an object.

Рендеринг блогу та постів

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.
- Покажемо приклад з Jinja2 template tool.
- See https://pypi.python.org/pypi/Jinja2.

```
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 keyword arguments.

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, яка має такий that title within the document.

- Заголовки блогів використовувались as sections and links within the index.
- Заголовки повинні бути унікальними, інакше отримаємо RST rendering errors.
- 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.

Dumping and loading with 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 C-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.

- there's a mapping to JavaScript types that JSON uses.

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

- Інші типи не підтримуються і повинні зводитись до одного з представлених вище за допомогою extension functions, які можна plug into функції dump() і load().

transforming our microblog objects into simpler Python lists and dicts

- 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.
- There are some values in the object's __dict__ that we'd rather not persist, such as the underlined text for a Post.
- Для кодування наших об'єктів у JSON необхідно надати функцію, яка спростить об'єкти до примітивних типів Python. Вона називається ***функцією за замовчуванням (default function)*** і постачає 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 relevant для Python.

Для декодування об'єкта із JSON-словника можна шукати ключ "__jsonclass__" як підказку, що один з наших класів потрібно to be built, not a built-in Python object.

- Назва класу може відображатись на об'єкт класу, а послідовність аргументів може використовуватись для конструювання екземпляру.
- Інші потужні JSON-кодувальники (зокрема з Django Web framework), 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 keyword-аргументів.

```
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)
```

Якщо обробити клас неможливо, викликається кодування за замовчуванням для існуючого кодувальника.

- This will handle the built-in classes.

text= json.dumps(travel, indent=4, default=blog_encode)

Передаємо нашу функцію blog_encode() в якості default= keyword parameter у функцію json.dumps().

- Вона використовується кодувальником JSON, щоб визначити кодування об'єкта.


```
{
    "__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 працювати всередині the structure of a JSON parsing.

- Об'єкти нашого класу кодувались у вигляді словників.
- Кожен dict, отриманий від декодера JSON ***може*** бути одним з наших класів.

The JSON decoder "object hook" – 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.

- If the three keys are present, then the given function is called with the arguments and keywords.
- We can use this object hook to parse a JSON object as follows:
- blog_data= json.loads(text, object_hook= blog_decode)
- This will decode a block of text, encoded in a JSON notation, using our blog_decode() function to transform dict into proper Blog and Post objects.

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 tweaked by a Man In The Middle attack.

- У цьому випадку JSON-документ is doctored у процесі проходження через веб-інтерфейс, який includes an untrustworthy server acting as a proxy.
- SSL – поширений метод для уникання проблеми.

За необхідності можна замінити eval() на словник that maps from name to class.

- We can change eval(some_dict['__class__']) to {"Post":Post, "Blog":Blog,"datetime.datetime":datetime.datetime: }[some_dict['__class__']]

This will prevent problems in the event that a JSON document is passed through a non-SSL-encoded connection. It also leads to a maintenance requirement to tweak this mapping each time the application design changes.

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

```
@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__= [])
```

Спростимо кодувальник (encoder).

```
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

Для кращої сумісності з іншими мовами we should properly encode the datetime object in a standard string and parse a standard string.

- Consider this small change to the encoding:

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

- The encoded output names the static method datetime.datetime.strptime() and provides the argument encoded datetime as well as the format to be used to decode it.

```
{
    "__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__": {}
        }
    }
}
```

Вивід для посту

Тепер маємо ISO-сформатовану дату замість індивідуальних полів.

- Відмовились від створення об'єкта через назву класу.

Значення __class__ розширено до назви класу або статичного методу.

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 використовується схожий підхід:

```
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__":{}}
```

Модуль pickle виконує нативне постійне зберігання об'єктів

Модуль pickle може перетворити складний об'єкт у байтовий потік, і навпаки.

- Найбільш очевидна річ, яку можна зробити з байтовими потоками – записати їх у файл, проте також поширена відправка їх по мережі або збереження в базу даних.
- Це не формат обміну даними, як JSON, YAML, CSV або XML.

Модуль pickle тісно інтегрований з Python багатьма шляхами.

- Наприклад, методи __reduce__() та __reduce_ex__() класу існують для підтримки pickle processing.

Піклинг нашого мікроблогу

```
import pickle

with open("travel_blog.p","wb") as target:

    pickle.dump( travel, target )
```

Виконує експорт всього об’єкта travel у заданий файл.

- Файл записано як raw bytes, тому функція open() використовує режим "wb".

Можна відновити picked object так:

```
with open("travel_blog.p","rb") as source:
    copy= pickle.load( source )
```

Проєктування класу для надійного pickle processing

Метод __init__() класу не використовується to unpickle об'єкта.

- Метод __init__() обходиться шляхом використання методу __new__() та setting the pickled values напряму в __dict__ об'єкта.

Це важливо, коли оголошення класу включає деяку обробку в __init__().

- Наприклад, якщо __init__() відкриває зовнішні файли, створює деяку частину GUI-інтерфейсу або виконує певне зовнішнє оновлення БД, тоді this will not be performed during unpickling.

If we compute a new instance variable during the __init__() processing, there is no real problem.

- For example, consider a Blackjack Hand object that computes the total of the Card instances when the Hand is created.
- The ordinary pickle processing will preserve this computed instance variable.
- It won't be recomputed when the object is unpickled.
- The previously computed value will simply be unpickled.

Клас з обробкою даних в методі __init__() повинен make special arrangements, щоб переконатись в коректності початкового процесингу даних

Можемо зробити 2 речі:

- Замість негайного startup processing в __init__() виконувати одноразову initialization processing.
  - Наприклад, операції із зовнішнім файлом повинні відкладатись до вимоги.
- Визначити методи __getstate__() і __setstate__(), які pickle використає, щоб зберігати та відновлювати стан об'єкта.
  - Метод __setstate__() може потім викликати той же метод, що __init__() викликає для одноразової initialization processing у звичайному Python-коді.

Приклад: початкові екземпляри Card, роздані в Hand are logged for audit purposes by the __init__() method

Версія Hand, яка нормально не працює під час unpickling:

```
class Hand_x:
    def __init__( self, dealer_card, *cards ):
        self.dealer_card= dealer_card
        self.cards= list(cards)
        for c in self.cards:
            audit_log.info( "Initial %s", c )
    def append( self, card ):
        self.cards.append( card )
        audit_log.info( "Hit %s", card )
    def __str__( self ):
        cards= ", ".join( map(str,self.cards) )
        return "{self.dealer_card} | {cards}".format( self=self,
cards=cards )
```

Присутні 2 logging locations

В методах __init__() та append().

- Обробка в __init__() працює неузгоджено між початковим створенням об'єкту та unpickling для його повторного відтворення.
- Here's the logging setup to see this problem:

```
import logging,sys
audit_log= logging.getLogger( "audit" )
logging.basicConfig(stream=sys.stderr, level=logging.INFO)
```

- This setup створює лог та перевіряє, щоб logging level був підходящим для перегляду audit information.
- Невеликий скрипт, що builds, pickles, and unpickles Hand:

```
h = Hand_x( FaceCard('K','♦'), AceCard('A','♣'), Card('9','♥') )
data = pickle.dumps( h )
h2 = pickle.loads( data )
```

При виконанні скрипту log entries, що записані під час роботи __init__(), не виписано при unpickling Hand.

Для того, щоб ретельно записати лог аудиту для unpickling, можна внести lazy logging tests по всьому класу.

- Наприклад, розширити __getattribute__() to write the initial log entries whenever any attribute is requested from this class.
- Це призведе до stateful logging and an if statement that is executed every time a hand object does something.

Краще рішення - to tap into the way state is saved and recovered by pickle.

```
class Hand2:
    def __init__( self, dealer_card, *cards ):
        self.dealer_card= dealer_card
        self.cards= list(cards)
        for c in self.cards:
            audit_log.info( "Initial %s", c )
    def append( self, card ):
        self.cards.append( card )
        audit_log.info( "Hit %s", card )
    def __str__( self ):
        cards= ", ".join( map(str,self.cards) )
        return "{self.dealer_card} | {cards}".format( self=self,
cards=cards )
    def __getstate__( self ):
        return self.__dict__
    def __setstate__( self, state ):
        self.__dict__.update(state)
        for c in self.cards:
            audit_log.info( "Initial (unpickle) %s", c )
```

Метод __getstate__() використовується під час піклингу, щоб зібрати дані про поточний стан об'єкта.

- Цей метод може повернути будь-що.
- У випадку об'єктів, які мають internal memoization caches, наприклад, кеш може не be pickled, щоб зберегти час та простір.
- Ця реалізація використовує внутрішній __dict__ без змін.

Метод __setstate__() використовується при unpickling, щоб reset значення об'єкту.

- Ця версія merges стан у внутрішню змінну __dict__, а потім записує підходящі logging entries.

Проблеми при роботі

Під час unpickling глобальні імена в pickle-потоці можуть вести до виконання довільного коду.

- Глобальні імена загалом є назвами класів або функцій.
- Проте є можливість включити глобальне ім'я, яке буде функцією з модулів (на зразок os або subprocess).
- Це дозволить атакувати додатки, які спробують передати pickled-об'єкти по Інтернету без strong SSL controls in place.
- Для локальних файлів такої проблеми немає.

Щоб уникнути виконання довільного коду, необхідно успадковувати від класу pickle.Unpickler.

- Переозначимо метод find_class(), щоб method to replace it with something more secure.

Потрібно враховувати кілька проблем анпіклінгу:

- Необхідно запобігти використанню вбудованих функцій exec() та eval().
- Необхідно запобігти використанню модулів та пакетів, які можуть бути unsafe. Наприклад, sys та os краще be prohibited.
- We have to permit the use of our application modules.

Приклад, який накладає деякі обмеження

```
import builtins
class RestrictedUnpickler(pickle.Unpickler):
    def find_class(self, module, name):
        if module == "builtins":
            if name not in ("exec", "eval"):
                return getattr(builtins, name)
        elif module == "__main__":
            return globals()[name]
        # elif module in any of our application modules...
        raise pickle.UnpicklingError(
        "global '{module}.{name}' is forbidden".format(module=module,
name=name))
```

Ця версія класу Unpickler допоможе уникнути великої кількості потенційних програм, that could stem from a pickle stream that was doctored.

- It permits the use of any built-in function except exec() and eval().
- It permits the use of classes defned only in __main__.
- In all other cases, it raises an exception

Модуль csv

Кодує та декодує прості екзмепляри списку або словника в CSV-нотацію.

- Як і для модуля json, це неповне рішення для постійного зберігання.
- Проте широке розповсюдження CSV файлів означає можливу потребу в конвертації між Python-об'єктами та CSV.

Робота з CSV-файлами включає ручне відображення об'єктів у CSV-структури.

- Потрібно ретельно спроєктувати відображення, remaining cognizant of the limitations of the CSV notation.
- This can be diffcult because of the mismatch between the expressive powers of objects and the tabular structure of a CSV fle.

Вміст кожного стовпця CSV-файлу – текст

При завантаженні даних з CSV-файлу необхідно конвертувати ці значення в більш корисні типи в додатках.

- Перетворення може ускладнитись by the way spreadsheets perform unexpected type coercion.
- Наприклад, можемо мати spreadsheet, де поштові індекси США змінюються на дробові числа табличним процесором.
- Коли таблиця (spreadsheet) зберігається в CSV, поштові індекси можуть стати дивними числовими значеннями.

Тому можемо застосувати перетворення such as ('ooooo'+row['zip'])[-5:] для відновлення leading zeroes.

- Інший сценарій – something such as "{0:05.0f}".format(float(row['zip'])) для відновлення leading zeroes.
- Також не забувайте, що файл може мати суміш поштових індексів ZIP та ZIP+4.

Подальше ускладнення роботи з CSV-файлами – їх часте заповнення вручну та з помилками чи скороченнями.

- Маючи відносно прості класи, можемо перетворити кожний instance у прості рядки (flat row) даних.
- Часто namedtuple – хороший відповідник для CSV-файлу та Python-об'єктів.
- Going the other way, we might need to design our Python classes around namedtuples if our application will save data in the CSV notation.

Для класів-контейнерів часто важко визначити представлення структурованих контейнерів у плоских рядках CSVфайлів.

- Хорошого вирішення таких невідповідностей немає, потрібно ретельно проєктувати перетворення даних.

Dumping simple sequences to CSV

Розглянемо відображення між екземплярами namedtuple та рядками в CSV-файлі на прикладі Python-класу:

```
from collections import namedtuple
GameStat = namedtuple( "GameStat", "player,bet,rounds,final" )
```

- Об'єкти визначимо як просту послідовність атрибутів.
- Заповнення таких об'єктів може виглядати так:

```
def gamestat_iter( player, betting, limit=100 ):
    for sample in range(30):
        b = Blackjack( player(), betting() )
        b.until_broke_or_rounds(limit)
        yield GameStat( player.__name__, betting.__name__, b.rounds,
b.betting.stake )
```

Запишемо дані в файл для подальшого аналізу

```
import csv
with open("blackjack.stats","w",newline="") as target:
    writer= csv.DictWriter( target, GameStat._fields )
    writer.writeheader()
    for gamestat in gamestat_iter( Player_Strategy_1, Martingale_Bet
):
        writer.writerow( gamestat._asdict() )
```

Існує 3 кроки для створення CSV writer:

- 1. Відкрити файл, задавши параметр newline = "" (можлива підтримка нестандартного завершення рядків для CSV-файлів).
- 2. Створити CSV writer-об'єкт (тут – екземпляр DictWriter).
- 3. Поставити заголовок на перший рядок файлу, що трохи спростить обмін даними, надаючи підказку про вміст CSV-файлу.

Як тільки writer-об'єкт підготовано, застосуємо його метод writerow() для запису кожного словника в CSV-файл.

- Деяке спрощення може дати метод writerows(), який очікує ітератор, а не окремий запис (row):

```
data = gamestat_iter( Player_Strategy_1, Martingale_Bet )
with open("blackjack.stats","w",newline="") as target:
    writer= csv.DictWriter( target, GameStat._fields )
    writer.writeheader()
    writer.writerows( g._asdict() for g in data )
```

Loading simple sequences from CSV

We can load simple sequential objects from a CSV fle with a loop

```
with open("blackjack.stats","r",newline="") as source:
    reader= csv.DictReader( source )
    for gs in ( GameStat(**r) for r in reader ):
        print( gs )
```

We've defned a reader object for our fle.

- As we know that our fle has a proper heading, we can use DictReader.
- This will use the frst row to defne the attribute names.
- We can now construct the GameStat objects from the rows in the CSV fle.
- We've used a generator expression to build rows.

In this case, we've assumed that the column names match the attribute names of our GameStat class defnition.

- We can, if necessary, confrm that the fle matches the expected format by comparing reader.fieldnames with GameStat._fields.
- As the order doesn't have to match, we need to transform each list of feld names into a set.
- Here's how we can check the column names:

assert set(reader.fieldnames) == set(GameStat._fields)

We've ignored the data types of the values that were read from the fle.

- The two numeric columns will wind up being string values when we read from the CSV fle.
- Because of this, we need a more sophisticated row-by-row transformation to create proper data values.
- Here's a typical factory function that performs the required conversions:

```
def gamestat_iter(iterator):
    for row in iterator:
        yield GameStat( row['player'], row['bet'], int(row['rounds']),
int(row['final']) )
```

We've applied the int function to the columns that are supposed to have numeric values.

- In the rare event where the fle has the proper headers but improper data, we'll get an ordinary ValueError from a failed int() function.
- We can use this generator function as follows:

```
with open("blackjack.stats","r",newline="") as source:
    reader= csv.DictReader( source )
    assert set(reader.fieldnames) == set(GameStat._fields)
    for gs in gamestat_iter(reader):
        print( gs )
```

- This version of the reader has properly reconstructed the GameStat objects by performing conversions on the numeric values.

Handling containers and complex classes

When we look back at our microblog example, we have a Blog object that contains many Post instances.

- We designed Blog as a wrapper around list, so that the Blog would contain a collection.
- When working with a CSV representation, we have to design a mapping from a complex structure to a tabular representation.

We have three common solutions:

- We can create two fles: a blog fle and a posting fle. The blog fle has only the Blog instances. Each Blog has a title in our example. Each Post row can then have a reference to the Blog row to which the posting belongs. We need to add a key for each Blog. Each Post would then have a foreign key reference to the Blog key.
- We can create two kinds of rows in a single fle. We will have the Blog rows and Post rows. Our writers entangle the various types of data; our readers must disentangle the types of data.
- We can perform a relational database join between the various kinds of rows, repeating the Blog parent information on each Post child.

There's no best solution among these choices.

We have to design a solution to the impedance mismatch between flat CSV rows and more structured Python objects.

The use cases for the data will defne some of the advantages and disadvantages.

- Creating two fles requires that we create some kind of unique identifer for each Blog so that a Post can properly refer to the Blog.
- We can't easily use the Python internal ID, as these are not guaranteed to be consistent each time Python runs.

A common assumption is that the Blog title is a unique key; as this is an attribute of Blog, it is called a natural primary key.

- This rarely works out well; we cannot change a Blog title without also updating all of the Posts that refer to the Blog.
- A better plan is to invent a unique identifer and update the class design to include that identifer.
- This is called a ***surrogate key.***
- The Python uuid module can provide unique identifers for this purpose.

Dumping and loading multiple row types in a CSV fle

Creating multiple kinds of rows in a single fle makes the format a bit more complex.

- The column titles must become a union of all the available column titles.
- Because of the possibility of name clashes between the various row types, we can either access rows by position—preventing us from simply using csv.DictReader—or we must invent a more sophisticated column title that combines class and attribute names.

The process is simpler if we provide each row with an extra column that acts as a class discriminator.

- This extra column shows us what type of object the row represents.
- The object's class name would work out well for this.

We might write blogs and posts to a single CSV fle using two different row formats

```
with open("blog.csv","w",newline="") as target:
    wtr.writerow(['__class__','title','date','title','rst_text','tags'])
    wtr= csv.writer( target )
    for b in blogs:
        wtr.writerow(['Blog',b.title,None,None,None,None])
        for p in b.entries:
            wtr.writerow(['Post',None,p.date,p.title,p.rst_text,p.tags])
```

We did not make the titles unique, so we can't use a dictionary reader.

- When allocating columns by position like this, each row allocates unused columns based on the other types of rows with which it must coexist.
- These additional columns are flled with None.
- As the number of distinct row types grows, keeping track of the various positional column assignments can become challenging.

the individual data type conversions can be somewhat baffling.

In particular, we've ignored the data type of the timestamp and tags.

- We can try to reassemble our Blogs and Posts by examining the row discriminators:

```
with open("blog.csv","r",newline="") as source:
    rdr= csv.reader( source )
    header= next(rdr)
    assert header == ['__class__','title','date','title','rst_
text','tags']
    blogs = []
    for r in rdr:
        if r[0] == 'Blog':
            blog= Blog( *r[1:2] )
            blogs.append( blog )
        if r[0] == 'Post':
            post= post_builder( r )
            blogs[-1].append( post )
```

We've used two assumptions about the columns in the CSV fle that has the same order and type as the parameters of the class constructors.

- For Blog objects, we used blog= Blog(*r[1:2]) because the one-and-only column is text, which matches the class constructor.
- When working with externally supplied data, this assumption might prove to be invalid.

To build the Post instances, we've used a separate function to map from columns to class constructor.

```
import ast
def builder( row ):
    return Post(
        date=datetime.datetime.strptime(row[2], "%Y-%m-%d %H:%M:%S"),
        title=row[3],
        rst_text=row[4],
        tags=ast.literal_eval(row[5]) )
```

This will properly build a Post instance from a row of text.

It converts the text for datetime and the text for the tags to their proper Python types.

- This has the advantage of making the mapping explicit.

In this example, we're using ast.literal_eval() to decode more complex Python literal values.

- This allows the CSV data to include a tuple of string values:
- "('#RedRanger', '#Whitby42', '#ICW')".

Filtering CSV rows with an iterator

```
def blog_iter(source):
    rdr= csv.reader( source )
    header= next(rdr)
    assert header == ['__class__','title','date','title','rst_
text','tags']
    blog= None
    for r in rdr:
        if r[0] == 'Blog':
            if blog:
                yield blog
            blog= Blog( *r[1:2] )
        if r[0] == 'Post':
            post= post_builder( r )
            blog.append( post )
    if blog:
        yield blog
```

We can refactor the previous load example to iterate through the Blog objects rather than constructing a list of the Blog objects.

- This allows us to skim through a large CSV fle and locate just the relevant Blog and Post rows.
- This function is a generator that yields each individual Blog instance separately

This blog_iter() function creates the Blog object and appends the Post objects.

- Each time a Blog header appears, the previous Blog is complete and can be yielded.
- At the end, the final Blog object must also be yielded.
- If we want the large list of Blog instances, we can use the following code:

```
with open("blog.csv","r",newline="") as source:
    blogs= list( blog_iter(source) )
```

This will use the iterator to build a list of Blogs in the rare cases that we actually want the entire sequence in memory.

We can use the following to process each Blog individually, rendering it to create RST fles

```
with open("blog.csv","r",newline="") as source:
    for b in blog_iter(source):
        with open(blog.title+'.rst','w') as rst_file:
            render( blog, rst_file )
```

- We used the blog_iter() function to read each blog.
- After being read, it can be rendered to an RST-format fle.
- A separate process can run rst2html.py to convert each blog to HTML.

We can easily add a flter to process only selected Blog instances.

- Rather than simply rendering all the Blog instances, we can add an if statement to decide which Blogs should be rendered.

Dumping and loading joined rows in a CSV fle

Joining the objects together means that each row is a child object, joined with all of the parent objects that child.

- This leads to repetition of the parent object's attributes for each child object.
- When there are multiple levels of containers, this can lead to large amounts of repeated data.

The advantage of this repetition is that each row stands alone and doesn't belong to a context defined by the rows above it.

- We don't need a class discriminator as parent values are repeated for each child object.

This works well for data that forms a simple hierarchy; each child has some parent attributes added to it.

- When the data involves more complex relationships, the simplistic parent-child pattern breaks down.
- In these examples, we've lumped the Post tags into a single column of text.
- If we tried to break the tags into separate columns, they would become children of each Post, meaning that the text of Post might be repeated for each tag.
- Clearly, this isn't a good idea!

The column titles must become a union of all the available column titles.

- Because of the possibility of name clashes between the various row types, we'll qualify each column name with the class name.
- This will lead to column titles such as 'Blog.title' and 'Post.title', which prevents name clashes.
- This allows for the use of DictReader and DictWriter rather than the positional assignment of the columns.
- However, these qualifed names don't trivially match the attribute names of the class defnitions; this leads to somewhat more text processing to parse the column titles.
- Here's how we can write a joined row that contains parent as well as child attributes:

```
with open("blog.csv","w",newline="") as target:
    wtr= csv.writer( target )
    wtr.writerow(['Blog.title','Post.date','Post.title', 'Post.
tags','Post.rst_text'])
    for b in blogs:
        for p in b.entries:
            wtr.writerow([b.title,p.date,p.title,p.tags,p.rst_text])
```

```
def blog_iter2( source ):
    rdr= csv.DictReader( source )
    assert set(rdr.fieldnames) == set(['Blog.title','Post.date','Post.
title', 'Post.tags','Post.rst_text'])
    row= next(rdr)
    blog= Blog(row['Blog.title'])
    post= post_builder5( row )
    blog.append( post )
    for row in rdr:
        if row['Blog.title'] != blog.title:
            yield blog
            blog= Blog( row['Blog.title'] )
        post= post_builder5( row )
        blog.append( post )
    yield blog
```

We saw qualifed column titles.

- In this format, each row now contains a union of the Blog attribute and the Post attributes.
- This is somewhat easier to prepare, as there's no need to fill unused columns with None.
- As each column name is unique, we could easily switch to a DictWriter too.
- Here's a way to reconstruct the original container from the CSV rows:

The first row of data is used to build a Blog instance and the first Post in that Blog.

- The invariant condition for the loop that follows assumes that there's a proper Blog object.
- Having a valid Blog instance makes the processing logic much simpler.
- The Post instances are built with the following function:

```
import ast
def post_builder5( row ):
    return Post(
        date=datetime.datetime.strptime(
            row['Post.date'], "%Y-%m-%d %H:%M:%S"),
        title=row['Post.title'],
        rst_text=row['Post.rst_text'],
        tags=ast.literal_eval(row['Post.tags']) )
```

We might want to refactor the Blog builder to a separate function

However, it's so small that adherence to the DRY principle seems a bit fussy.

- Because the column titles match the parameter names, we might try to use something like the following code to build each object:

```
def make_obj( row, class_=Post, prefix="Post" ):
    column_split = ( (k,)+tuple(k.split('.')) for k in row )
    kw_args = dict( (attr,row[key])
        for key,classname,attr in column_split if
classname==prefix )
    return class( **kw_args )
```

We used two generator expressions here.

- The frst generator expression splits the column names into the class and attribute and builds a 3-tuple with the full key, the class name, and the attribute name.
- The second generator expression flters the class for the desired target class; it builds a sequence of 2-tuples with the attribute and value pairs that can be used to build a dictionary.

This doesn't handle the data conversion for Posts.

The individual column mappings simply don't generalize well.

- Adding lots of processing logic to this isn't very helpful when we compare it to the post_builder5() function.

In the unlikely event that we have an empty fle—one with a header row but zero Blog entries—the initial row=next(rdr) function will raise a StopIteration exception.

- As this generator function doesn't handle the exception, it will propagate to the loop that evaluated blog_iter2(); this loop will be terminated properly.