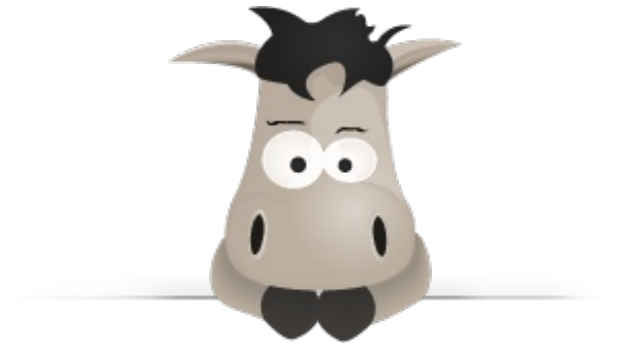

Sérialisez vos objets au format JSON !

Par Romain Porte (MicroJoe)

www.siteduzero.com

Dernière mise à jour le 14/04/2012

Sommaire

Sérialisez vos objets au format JSON !

Par Romain Porte (MicroJoe)

Mise à jour : 14/04/2012
Difficulté : Facile Durée d'étude : 1 heure

Ah, la POO… Quel plaisir de créer et de manipuler des objets aussi facilement ! Mais comment les sauvegarder simplement ou les faire transiter sur un réseau ?
En utilisant la sérialisation, pardi !

Dans le tutoriel officiel de ce site concernant Python, les auteurs abordent l'utilisation du module *pickle* pour mener à bien la sérialisation de vos classes.
Je tiens à vous présenter le module JSON qui a le même objectif : sauvegarder et restaurer les attributs de vos classes.

Tout au long de ce tutoriel, on distinguera bien **le format de fichier JSON** représentant la manière dont laquelle sont organisées les données dans le fichier et **le module Python `json`** qui permet, quant à lui, de manipuler cette représentation de données.
Sommaire du tutoriel :

Présentation du format JSON

Dans cette partie, je vais vous présenter le format JSON et ses spécificités.

Introduction au format JSON

Citation : Wikipédia

> JSON (*JavaScript Object Notation*) est un format de données textuel, générique, dérivé de la notation des objets du langage ECMAScript. Il permet de représenter de l'information structurée.

Euh… J'ai lu « *JavaScript Object Notation* ». Pourquoi venir nous parler de JavaScript dans un tutoriel concernant Python ?
C'est quoi cette histoire ?

En fait, JSON est un format permettant de représenter des données. On peut comparer son usage à celui du XML.

Tout comme le XML, le JSON peut être lu par de nombreux langages de programmation (33 à l'heure actuelle) dont le Python et même… l'humain !
En effet, il s'agit d'un format texte et contrairement au format binaire (**utilisé par *pickle***), vous pourrez voir et modifier facilement des valeurs avec un quelconque éditeur de texte.

Présentation du format JSON par l'exemple

Je vais ici vous présenter deux exemples de fichiers représentant les caractéristiques d'une *playlist* : l'un en XML et l'autre en JSON.
Voici sa représentation « humaine ».

La *playlist* nommée **MeshowRandom** est composée de :

- ***Best Improvisation Ever 2*** de **David Meshow** avec une note de **5/5** ;
- ***My Theory*** de **David Meshow** avec une note de **4/5**.

Voyons maintenant sa représentation « informatique » en examinant les fichiers correspondants à cette description.

Tout d'abord, en XML !

Le format de balisage XML est très répandu (le zCode est dérivé du XML !).

Code : XML

```xml
<?xml version="1.0" ?>
<playlist nom="MeshowRandom">
    <chanson>
        <titre>Best Improvisation Ever 2</titre>
        <auteur>David Meshow</auteur>
        <note>5</note>
    </chanson>
    <chanson>
        <titre>My Theory (Bonus)</titre>
        <auteur>David Meshow</auteur>
        <note>4</note>
    </chanson>
</playlist>
```

Et en JSON, ça donne quoi ?

Voici les mêmes informations contenues dans un fichier JSON.

Code : JavaScript

```javascript
{
    "nom" : "MeshowRandom",

    "chansons" : [
        {
            "titre" : "Best Improvisation Ever 2",
            "auteur" : "David Meshow",
            "note" : 5
        },
        {
            "titre" : "My Theory",
            "auteur" : "David Meshow",
            "note" : 4
        }
    ]
}
```

Bilan

À partir de cette comparaison, on peut clairement voir émerger l'avantage principal du format JSON par rapport au format XML : **il est minimaliste**.

En effet, JSON se base plus sur un modèle clé/valeur que sur un format de balisage. Cela permet d'éviter les balises de

fermeture, la répétition et cela peut éventuellement représenter un gain de place sur de très gros fichiers (16 caractères économisés ici en comptant les espaces, soit 95% de la taille originale).

Attention cependant : le format JSON ne possède pas de spécification **concernant les commentaires**. Vous pouvez en rajouter de la même manière qu'en JavaScript avec `// Commentaire` ou `/* Commentaire */` mais rien ne garantit qu'ils seront pris en compte et ils pourraient (dans le pire des cas) faire planter votre programme selon votre parseur.
Dans le doute, **évitez les commentaires dans vos fichiers JSON** histoire de rester conforme à la norme officielle.

Présentation du module

Eh bien tout d'abord, comme tout module, il faut l'inclure. Pour ce faire, rien de plus simple :

Code : Python

```
import json
```

En important le module de cette manière, cela m'oblige à ajouter à chaque fois le préfixe `json` devant les fonctions du module.
Je trouve cela plus rigoureux ; mais libre à vous d'utiliser `from json import *` si cela vous dérange (même si c'est très déconseillé).

Sachez que ce module est déjà capable de sérialiser tous les types standards de Python hormis les *tuples* et les entrées binaires (en même temps, pour un format texte…).

Premier exemple : sérialiser dans une chaîne de caractères

Dans cet exemple, je vais tenter de reproduire la *playlist* que je vous ai montrée dans la partie de découverte du format JSON.
Cependant, pour plus de commodité, je vais seulement enregistrer les titres des musiques ; nous verrons dans l'annexe comment procéder avec une classe `Musique`.

Tout d'abord, on inclut le module JSON :

Code : Python Console

```
>>> import json
```

Ensuite, on peuple la *playlist* :

Code : Python Console

```
>>> playlist = {}
>>> playlist["nom"] = "MeshowRandom"
>>> playlist["musiques"] = []
>>> playlist["musiques"].append("Best Improvisation Ever 2")
>>> playlist["musiques"].append("My Theory (Bonus)")
>>>
>>> print(playlist)
{'musiques': ['Best Improvisation Ever 2', 'My Theory (Bonus)'],
'nom': 'MeshowRandom'}
```

Voici à présent la partie intéressante : on va demander au module de transposer notre dictionnaire au format JSON.
Pour cela, on va se servir de la fonction `json.dumps(objet)` (le *s* signifie ici *string*).

Code : Python Console

```
>>> print(json.dumps(playlist))
{"musiques": ["Best Improvisation Ever 2", "My Theory (Bonus)"],
"nom": "MeshowRandom"}
```

Vous pouvez constater que la fonction nous retourne une chaîne de caractères décrivant bien notre *playlist* au format JSON !

Mais le code est sur une seule ligne et n'est pas indenté…
C'est illisible ! 

En effet, ce n'est pas très lisible pour nous, mortels.
On peut heureusement procéder comme ceci pour indenter automatiquement la sortie de la fonction :

Code : Python Console

```
>>> print(json.dumps(playlist, indent=4))
{
"musiques": [
"Best Improvisation Ever 2",
"My Theory (Bonus)"
],
"nom": "MeshowRandom"
}
```

C'est déjà plus clair. 

Deuxième exemple : sérialiser dans un fichier

Eh bien, il suffit de sauvegarder la chaîne retournée par `json.dumps(objet)` dans un fichier, non ?
Pourquoi ce deuxième exemple ?

On pourrait le faire, mais il est plus pratique d'utiliser une fonction du module dédiée à l'écriture dans les fichiers : la fonction `json.dump(objet, flux)`.
Celle-ci va directement écrire dans le flux de données (ici notre fichier), sans passer par une chaîne de caractères intermédiaire.

On va conserver la *playlist* remplie précédemment et utiliser cette fonction :

Code : Python Console

```
>>> with open('Test.json', 'w', encoding='utf-8') as f:
...     json.dump(playlist, f, indent=4)
```

Voici ce que j'obtiens :

Code : JavaScript - Test.json

```
{
    "musiques": [
        "Best Improvisation Ever 2",
        "My Theory (Bonus)"
    ],
    "nom": "MeshowRandom"
}
```

C'est beau, n'est-ce pas ?

On remarquera que les clés ont été rangées un peu n'importe comment ; cela est dû au fait que Python range les paires de façon arbitraire.
Si vous voulez que les attributs soient sérialisés dans le même ordre que celui que vous avez indiqué, alors utilisez un `OrderedDict` fourni dans le module `collections`.

Voici un exemple :

Code : Python Console

```
>>> from collections import OrderedDict
>>> dct = OrderedDict()
>>> dct["__class__"] = "Playlist"
>>> dct["name"] = "MeshowRandom"
>>> dct["description"] = "Cool stuff."
>>>
>>> import json
>>> print(json.dumps(dct, indent="4"))
{
"__class__": "Playlist",
"name": "MeshowRandom",
"description": "Cool stuff."
}
```

Et après ?

Désormais, vous savez sérialiser les principaux objets de Python au format JSON, et ce, dans des fichiers.
Cependant, avant de vous quitter, j'aimerais aborder la sérialisation d'instances de classes inconnues du module `json` telles que la classe `Playlist` ou la classe `Musique`.

Pour cela, rendez-vous dans la partie suivante. 

Sérialisons nos propres classes !

Nous allons maintenant nous attaquer au vif du sujet : comment sérialiser nos classes avec le module `json`.
Prenons cette classe comme exemple : la classe *Playlist* (encore et toujours ).

Code : Python

```
class Playlist:
    def __init__(self, nom):
        nom = nom
        musiques = []
```

Dans cet exemple, les entrées dans `musiques` seront de simples chaînes de caractères représentant les titres des chansons.
Libre à vous de les remplacer par des dictionnaires ou, mieux, des instances de la classe Musique qui contient son auteur, sa note, etc. (Il faudra néanmoins que la classe Musique soit également sérialisable en JSON pour pouvoir sérialiser la *playlist*).

Sérialisation

Voici la fonction qui va sérialiser notre objet :

Code : Python

```
def serialiseur_perso(obj):
```

```
        if isinstance(obj, Playlist):
            return {"__class__": "Playlist"
                    "nom": obj.nom,
                    "musiques": obj.musiques}
        raise TypeError(repr(obj) + " n'est pas sérialisable !")
```

Je vais vous décrire pas à pas ce que produit ce code :

- d'abord, on vérifie que l'objet passé est bien de type *Playlist* ;
- si l'objet est bien une instance de la classe *Playlist*, **on retourne un type sérialisable par le module** (ici un dictionnaire) contenant les attributs à sauvegarder de notre classe ;
- sinon, on lance une exception disant que l'objet passé n'est pas sérialisable.

Ingénieux, non ?

Euh… c'est quoi, l'entrée `"__class__"` dans le dictionnaire ?
Ça se mange ?

Il s'agit en fait d'une entrée qui n'est pas contenue dans l'objet en lui-même mais elle va nous permettre de savoir de quel type est la représentation JSON de l'objet. Vous comprendrez mieux son utilité lors de la désérialisation.

Désérialisation

Nous allons maintenant faire l'inverse de la fonction `serialiser_json` : convertir le dictionnaire obtenu par le module json en *playlist* :

Code : Python

```
def deserialiseur_perso(obj_dict):
    if "__class__" in obj_dict:
        if obj_dict["__class__"] == "Playlist":
            obj = Playlist(objet["nom"])
            obj.musiques = objet["musiques"]
            return obj
    return objet
```

Voici ce que fait le code :

- on regarde si `"__class__"` est défini dans le dictionnaire `obj_dict` ;
- si oui, on vérifie si `"__class__" == "Playlist"` et on retourne l'objet `obj` créé à l'aide du contenu du dictionnaire ;
- sinon, on retourne le dictionnaire.

Utilisation

Sérialiser un objet

Code : Python

```
# On crée un objet de type Playlist et on le peuple.
playlist = Playlist("MeshowRandom")
playlist.musiques.append("Best improvisation ever 2")
playlist.musiques.append("My Theory")

# On l'enregistre dans un fichier JSON avec notre sérialiseur
perso.
```

```
with open("MaPlaylist.json", "w", encoding="utf-8") as fichier:
json.dump(playlist, fichier, default=serialiseur_perso)
```

Désérialiser un objet

Code : Python

```
# On crée un objet vide.
playlist = Playlist("untitled")

# On le peuple à l'aide du fichier JSON.
with open("MaPlaylist.json", "r", encoding="utf-8") as fichier:
playlist = json.load(fichier, object_hook=deserialiseur_perso)
```

Ça y est, vous êtes désormais capable de sérialiser toutes vos classes dans des fichiers JSON !

Annexe

Sérialiser plusieurs objets

Contexte

Supposons que vous ne vouliez pas sérialiser une seule *playlist* mais plusieurs situées dans un tableau.
Eh bien, étant donné que les tableaux Python sont sérialisables par défaut avec le module, il ne devrait pas y avoir de problème !

Résultat

On peuple deux *playlists* et on les ajoute dans un tableau puis on enregistre ce même tableau :

Code : Python

```
pl1 = Playlist("MeshowRandom")
pl1.musiques.append("Best improvisation ever 2")
pl1.musiques.append("My theory")

pl2 = Playlist("MeshowRandom2")
pl2.musiques.append("Corolla song")
pl2.musiques.append("Best guitar improvisation ever")

liste_playlists = []
liste_playlists.append(pl1)
liste_playlists.append(pl2)

with open('Test.json', 'w', encoding='utf-8') as f:
    json.dump(liste_playlists, f, indent=4,
default=serialiseur_perso)
```

Et voici ce que ça donne :

Code : JavaScript - Test.json

```
[
    {
        "musiques": [
            "Best improvisation ever 2",
            "My theory"
        ],
```

```
            "nom": "MeshowRandom",
            "__class__": "Playlist"
        },
        {
            "musiques": [
                "Corolla song",
                "Best guitar improvisation ever"
            ],
            "nom": "MeshowRandom2",
            "__class__": "Playlist"
        }
    ]
```

Sérialiser plusieurs classes différentes

Contexte

Supposons que vous vouliez sérialiser votre *playlist* qui contient elle-même des instances de la classe `Musique` :

Code : Python

```
class Playlist:
    def __init__(self, nom):
        self.nom = nom
        self.musiques = []

class Musique:
    def __init__(self, titre, auteur, note = 3):
        self.titre = titre
        self.auteur = auteur
        self.note = note
```

Il va donc falloir étendre la fonction `serialiseur_perso` en lui indiquant comment sérialiser la classe `Playlist` **et** la classe `Musique`.
Voici ce que ça donne :

Code : Python

```
def serialiseur_perso(obj):

    # Si c'est une musique.
    if isinstance(obj, Musique):
        return {"__class__": "Musique",
                "titre": obj.titre,
                "auteur": obj.auteur,
                "note": obj.note}

    # Si c'est une playlist.
    if isinstance(obj, Playlist):
        return {"__class__": "Playlist",
                "nom": obj.nom,
                "musiques": obj.musiques}

    # Sinon le type de l'objet est inconnu, on lance une exception.
    raise TypeError(repr(obj) + " n'est pas sérialisable !")
```

Résultat

Voici le code d'exemple, dans lequel on va peupler la *playlist* d'instances de la classe `Musique` :

Code : Python

```
playlist = Playlist("MeshowRandom")
playlist.musiques.append(Musique("Best Improvisation Ever 2", "David Meshow", 5))
playlist.musiques.append(Musique("My Theory", "David Meshow", 4))

with open('Test.json', 'w', encoding='utf-8') as f:
    json.dump(playlist, f, indent=4, default=serialiseur_perso)
```

Et enfin le contenu du fichier :

Code : JavaScript - Test.json

```
{
    "musiques": [
        {
            "note": 5,
            "titre": "Best Improvisation Ever 2",
            "auteur": "David Meshow",
            "__class__": "Musique"
        },
        {
            "note": 4,
            "titre": "My Theory",
            "auteur": "David Meshow",
            "__class__": "Musique"
        }
    ],
    "nom": "MeshowRandom",
    "__class__": "Playlist"
}
```

Utilisation avec l'héritage

Contexte

Supposons que certaines de vos *playlists* peuvent être imagées, voici alors ce que l'on pourrait avoir :

Code : Python

```
class Playlist:
    def __init__(self, nom):
        self.nom = nom
        self.musiques = []

class ImagedPlaylist(Playlist):
    def __init__(self, nom):
        Playlist.__init__(self, nom)
        self.image = ""
```

La classe `ImagedPlaylist` hérite donc de la classe `Playlist`, mais comment faire pour sérialiser le tout ?

Nous allons tout d'abord enregistrer les attributs de la classe `ImagedPlaylist` puis ensuite, dans un champ nommé `"__parent__"`, nous transformeront cet objet de type `ImagedPlaylist` en objet de type `Playlist` (*downcasting*) pour pouvoir enregistrer les attributs hérités de l'objet parent (comme l'attribut `musiques` par exemple).

Voici le code Python qui vous permettra d'effectuer un *downcasting* :

Code : Python

```
import copy

def ImagedPlaylist2Playlist(obj):
    obj_cpy = copy.copy(obj)           # On copie l'objet
    obj_cpy.__class__ = Playlist       # On change son attribut
__class__ pour le convertir

    return obj_cpy
```

Pour les plus sceptiques, voici la preuve en image :

Code : Python Console

```
>>> import copy
>>> obj = ImagedPlaylist("MeshowRandom")
>>>
>>> obj_cpy = copy.copy(obj)
>>> obj_cpy.__class__ = Playlist
>>>
>>> print(obj)
<__main__.ImagedPlaylist object at 0x0000000003380630>
>>> print(obj_cpy)
<__main__.Playlist object at 0x0000000003385DD8>
```

Ainsi nous allons pouvoir directement le rajouter dans notre fonction `serialiseur_perso` :

Code : Python

```
import copy
def serialiseur_perso(obj):
    if isinstance(obj, ImagedPlaylist):
obj_cpy = copy.copy(obj)
obj_cpy.__class__ = Playlist
        return {"__class__": "ImagedPlaylist",
                "image": obj.image,
"__parent__": obj_cpy}

    if isinstance(obj, Playlist):
        return {"__class__": "Playlist",
                "nom": obj.nom,
                "musiques": obj.musiques}
```

A la ligne 8, on va demander au module de sérialiser un objet de type Playlist, il va donc en interne rappeler cette fonction avec l'objet de type Playlist que nous venons de convertir !

Résultat

Le tout marche très bien, voici ce que l'on peut obtenir :

Code : JavaScript

```
{
    "image": "Test.png",
    "__class__": "ImagedPlaylist",
```

```
            "__parent__": {
                "musiques": [],
                "nom": "MeshowRandom",
                "__class__": "Playlist"
            }
        }
```

Q.C.M.

Le premier QCM de ce cours vous est offert en libre accès.
Pour accéder aux suivants

Connectez-vous Inscrivez-vous
Qu'est-ce que le format JSON ?

- ◯ Un format de playlist.
- ◯ Un format de données minimaliste.
- ◯ Un fichier se terminant par « .json ».

À quoi sert la sérialisation ?

- ◯ À communiquer sur le port série.
- ◯ À enregistrer des classes.
- ◯ À enregistrer des instances de classes.

Correction !

Statistiques de réponses au QCM

Nous voilà arrivés au terme de ce tutoriel. Vous pouvez à présent supprimer les *parseurs* faits maison de fichiers texte et utiliser le format JSON ainsi que son module Python sans modération.

Sources :

- JSON sur Wikipédia (pour la présentation du format JSON) ;
- *Serializing Python objects* tiré du fameux livre *Dive into Python 3* pour les fonctions `serialiseur_perso` et `deserialiseur_perso` ;
- et évidemment la doc du module !

Remerciements :

- Thunderseb pour ses conseils ;
- L'équipe des zCorrecteurs et plus particulièrement Guill@um€ et Stylla pour leur relecture attentive ;
- Et Zopieux pour ses précisions.

Ce tutoriel a été corrigé par les zCorrecteurs.