Python et Json

Vérification du JSON via JSON SCHEMA

Table des matières

JSON est un format souvent apprécié en Python, car il est nativement parlant, similaire à un dictionnaire pour notre langage adoré.

Souvent opposé au format XML, on lui reproche en général de ne pas avoir de système de validation du format. C'est dans cette optique qu'est né le « JSON SCHEMA ».

C'est ce « JSON SCHEMA » et son utilisation avec Python que je vous invite à découvrir dans cet article.

Commentez

Article lu -1 fois.

L'auteur

Alexandre GALODE

L'article

Publié le 11 novembre 2017

TOUT PUBLIC

Version PDF Version hors-ligne

ePub, Azw et Mobi

Liens sociaux

Partager

I. Introduction▲

Pendant JSON du XSD pour le XML, le JSON SCHEMA permet de définir la façon dont un JSON doit être structuré.

Il est ainsi aisé de s'assurer qu'un JSON est correctement formaté afin de le charger dans un traitement ou un logiciel.

Cependant, peu connu, le JSON SCHEMA reste encore à appréhender. À travers cet article, nous aborderons la structure des JSON SCHEMAS, et leur utilisation pour améliorer nos développements.

Cet article est écrit pour Python3.x. Des différences peuvent exister avec Python2.x.

La version de JSON_SCHEMA considérée dans cet article est la " Draft 4 ".

II. Le JSON SCHEMA▲

II-A. Type de données▲

Comme vous pouvez le constater, rien de compliqué.

Voici ainsi comment on définit qu'on attend un item de type integer ou float :

Sélectionnez

```
{ "type": "number"}
```

Cependant, on pourrait s'attendre à avoir différents types possibles. JSON SCHEMA prévoit ce cas via les listes :

Sélectionnez

```
{ "type": ["number", "string"]}
```

Ici, nous déclarons attendre au choix un nombre ou une chaîne de caractères.

Il existe également un type " integer ". Cependant, faisant partiellement doublon avec le type " number " et pouvant amener de la confusion, il n'est pas toujours pris en charge par les validateurs de JSON SCHEMA. C'est la raison pour laquelle il n'est pas abordé dans cet article.

II-B. Propriétés de données▲

En JSON_SCHEMA, un objet, ou " object ", est assimilable à un dictionnaire, possédant des propriétés. Les propriétés de cet objet sont elles-mêmes définies dans un dictionnaire.

Sélectionnez

```
{
    "type": "object",
    "properties":
    {
        "AGE": {"type": "number"},
        "NOM": {"type": "string"},
    }
}
```

Prenons ce JSON, par exemple. Ici, nous indiquons que nous attendons un objet devant contenir deux propriétés :

- un nom, qui doit être un nombre ;
- un âge qui doit être un nombre.

Les propriétés définies correspondent alors aux clés d'un dictionnaire Python. Les valeurs attendues constituent donc les items du dictionnaire.

Le JSON SCHEMA impose que les clés ne soient qu'au format string.

Pour revenir à notre JSON SCHEMA d'exemple, voici un JSON qui serait validé :

Sélectionnez

```
{
    "CANDIDAT01":
    {
        "AGE": 18,
        "NOM": "Elouann"
    }
}
```

De même, voici un JSON qui ne serait pas validé (âge au format String et non Number) :

Sélectionnez

```
{
    "CANDIDAT01":
    {
        "AGE": "18",
        "NOM": "Elouann"
    }
}
```

II-C. Entête de fichier▲

Bien que non obligatoire, il existe une bonne pratique consistant à ajouter une ligne au début de chaque JSON SCHEMA.

Le but est simplement de savoir, à l'ouverture d'un JSON, s'il s'agit ou non d'un JSON SCHEMA :

Sélectionnez

```
{
    "$schema": "http://json-schema.org/schema#"
```

II-D-1. Les mots clés génériques▲

Les mots clés qui suivent peuvent être utilisés pour n'importe quels types de données.

Mot clé	Type de valeur	Description
"title"	String (Courte de préférence)	Sert à nommer un JSON, en début de fichier
"description"	String	Permet de décrire l'utilité du JSON SCHEMA
"enum"	Liste	Permet de stipuler une liste de valeurs autorisées
"anyOf"	Liste	Au moins l'une des configurations déclarées doit être vérifiée.
"allOf"	Liste	Toutes les configurations déclarées doivent être vérifiées.
"oneOf"	Liste	Seule l'une des configurations déclarées doit être vérifiée.
"not"	Liste	Permet de déclarer quelque chose qui ne doit pas être présent.
"required"	Liste	Permet de définir une liste d'éléments obligatoires, clés au format string
"pattern"	String	Commençant par " ^ " et finissant par " $ ". Permet de stipuler une REGEX.

Voici un exemple de JSON SCHEMA mettant en œuvre ces mots clés.

Sélectionnez

```
{
    "$schema": "http://json-schema.org/schema#",
    "title": "JSON SCHEMA de demonstration",
    "description": "Ceci est un JSON SCHEMA pour le site Developpez",

    "type": "object",
    "oneOf": [{"required": ["AGE", "NAME", "TEL_FIXE"]},
              {"required": ["AGE", "NAME", "TEL_CELL"]}],
    "properties":{"AGE": {"type": "number"},
                  "NAME": {"type": "string"},
                  "FAVORITE_COLOR": {"type": "string",
                                     "enum": ["red", "blue", "yellow"]
                                    },
                  "PHONE": {"type": "string", "pattern":"^[0-9]{4}$"},
                  "CELL": {"type": "string", "pattern":"^[0-9]{4}$"}
                 }
```

- " AGE " est obligatoire et doit être de type " int " ou " float " ;
- " NAME " est obligatoire et doit être une chaîne de caractères ;
- " FAVORITE_COLOR " n'est pas obligatoire ; si elle est renseignée, elle ne peut être que " red ", " blue " ou " yellow " ;
- " TEL_FIXE " et " TEL_PORTABLE " doivent être des chaînes de caractères ; au moins l'une des deux doit exister dans le JSON testé.

Cet exemple permet de voir comment bien définir un JSON SCHEMA.

Sur les trois premières lignes, nous retrouvons la ligne informant qu'il s'agit d'un JSON SCHEMA, puis un titre et une description rapide.

Ensuite, nous définissons un dictionnaire. J'attire votre attention à cet endroit. Bien que non nécessaire, cette partie est extrêmement importante.

En effet, le fait de déclarer un dictionnaire va nous simplifier la vie par la suite. Nous sommes en train de définir notre fichier JSON en tant que dictionnaire Python.

L'avantage de cette solution est de pouvoir préciser facilement, dans les deux lignes qui suivent, les diverses combinaisons que nous autorisons, à savoir que le nom et l'âge sont obligatoires, la couleur préférée optionnelle, et qu'au moins un des deux numéros de téléphone doit être renseigné.

Voici un exemple de JSON valide avec ce JSON SCHEMA

Sélectionnez

```
1.
2.
3.
4.
5.
6.
{
    "AGE": 12,
    "NAME": "DVP",
    "FAVORITE_COLOR": "red",
    "PHONE": "0000"
}
```

Et un autre invalide. Nous verrons plus loin comment les tester. Vous aurez alors le loisir d'effectuer par vous-même les validations, ou non, des divers JSON de cet article.

Sélectionnez

```
1.
2.
3.
4.
5.
6.
{
    "AGE": 12,
    "NAME": "DVP",
    "FAVORITE_COLOR": "red",
    "PHONE": "+0000"
}
```

II-D-2. Les " booleans "▲

Peu de choses à dire sur les booléens, en JSON SCHEMA. Par rapport à Python, seule la casse change.

Équivalence de terme	
JSON	**Python**
false	False
true	True

II-D-3. Les " numbers "▲

Les mots clés spécifiques aux nombres sont relativement simples :

Mot clé	Type de valeur	Description
"minimum"	Int ou float	Permet de stipuler une valeur minimale attendue

		minimum. S'il est mis à false, on testera alors : valeur ≥ minimum
"maximum"	Int ou float	Permet de stipuler une valeur maximale attendue
"exclusiveMaximum"	Booléen	Similaire à "exclusiveMinimum"
"multipleOf"	Int ou float	Permet de stipuler qu'on attend un multiple de la valeur indiquée.

Si nous reprenons l'exemple précédent, nous pouvons interagir au niveau de l'âge.

Nous allons ainsi ajuster ce paramètre:

- âge minimum de 0 ;
- âge maximum de 130 ;
- le nombre doit être un entier (multiple de 1).

Sélectionnez

```
{
    "$schema": "http://json-schema.org/schema#",
    "title": "JSON SCHEMA de demonstration",
    "description": "Ceci est un JSON SCHEMA pour le site Developpez",
    "type": "object",
    "anyOf": [{"required": ["AGE", "NAME", "PHONE"]},
              {"required": ["AGE", "NAME", "PHONE_CELL"]}],
    "properties":{"AGE": {"type": "number",
                          "minimum": 0,
                          "maximum": 130,
                          "multipleOf": 1},
                  "NAME": {"type": "string"},
                  "FAVORITE_COLOR": {"type": "string",
                                     "enum": ["red", "blue", "yellow"]
                                    },
                  "PHONE": {"type": "string", "pattern":"^[0-9]{4}$"},
                  "CELL": {"type": "string", "pattern":"^[0-9]{4}$"}
                 }
}
```

II-D-4. Les " strings "▲

Une fois que vous avez déclaré attendre une entrée de type chaîne de caractères, vous pouvez alors préciser les éléments suivants :

Mot clé	Type de valeur	Description
"minLength"	Int	Longueur minimale attendue
"maxLength"	Int	Longueur maximale attendue

Par exemple :

Sélectionnez

```
{
    "type": "string",
    "minLength": 2,
    "maxLength": 8,
    "pattern": "^([0-9]{2-8})$"
}
```

Cependant, afin de vous simplifier la vie, il existe certains contrôles prédéfinis via le mot clé "format".

Mot clé	Type de valeur	Description
"date-time"	String	Vérifie la conformité d'une date selon la norme RFC3339
"email"	String	Vérifie la validité d'une adresse mail selon la norme RFC5322

		RFC1034
"ipv4"	String	Vérifie la validité d'une adresse IP V4 selon la norme RFC2673
"ipv6"	String	Vérifie la validité d'une adresse IP V6 selon la norme RFC2373
"uri"	String	Vérifie la validité d'une adresse réseau selon la norme RFC3986

Le mot clé " format " n'est que rarement pris en compte par les outils de JSON SCHEMA.

La librairie que nous verrons plus loin ne gère pas ce mot clé. Vous devez alors avoir recours à une REGEX.

II-D-5. Les " arrays "▲

Les " arrays " sont assez simples à paramétrer. Comme le montre le tableau ci-dessous, peu de mots clés disponibles.

Mot clé	Type de valeur	Description
"items"	---	Permet de définir un type global pour toute la liste. Possibilité de définir précisément le type de chaque item si longueur connue
"minItems"	Int	Nombre minimum d'items
"maxItems"	Int	Nombre maximum d'items
"uniqueItems"	Booléen	Autorise (False) ou non la présence de doublons dans une liste (unicité)

Par exemple, si on attend une liste de longueur comprise entre 2 et 4 éléments (compris), et sans doublons possibles, on écrira :

Sélectionnez
```
1.
2.
3.
4.
5.
6.
```
```
{
    "type": "array",
    "minLength": 2,
    "maxLength": 4,
    "uniqueItems": true
}
```

II-D-6. Les " objects "▲

Les « objects » JSON SCHEMA, équivalents des dictionnaires Python, sont un peu plus complexes à maîtriser. En effet, constituant la base des JSON SCHEMAS, ils sont plus complets que les types précédents, en termes de mots clés.

Mot clé	Type de valeur	Description
"minProperties"	Int	Nombre minimum de clés attendues
"maxProperties"	Int	Nombre maximum de clés attendues

"additionalProperties"	Booléen	Booléen. Autorise (True) ou non la présence de clés non définies dans le JSON SCHEMA
"additionalProperties"	Dictionnaire	Permet de définir les paramètres des éléments supplémentaires tels les types
"dependencies"	Liste	Permet de créer des relations entre clés (ex : clé 1 obligatoire si clé 2 présente). Non réciproque, attention.

Pour expliciter un peu ces divers mots clés, je vous propose quelques JSON SCHEMA d'exemples.

Sélectionnez

```
{
  "type": "object",
  "minProperties": 2,
  "maxProperties": 3
}
```

Sélectionnez

```
{
  "type": "object",
  "properties":{"name": {"type": "string"},
                "town": {"type": "string"},
                "zip": {"type": "number"}
               },
  "dependencies":{"zip": ["town"],
                  "town": ["zip"]}
}
```

Sélectionnez

```
{
  "type": "object",
  "properties":{"name": {"type": "string"},
                "town": {"type": "string"},
                "zip": {"type": "number"}
               },
  "additionalProperties":{"type": ["string", "number"]}
}
```

III. Mise en pratique▲

Maintenant que nous avons passé en revue le fonctionnement global du JSON SCHEMA, je vous invite à le mettre en pratique avec Python.

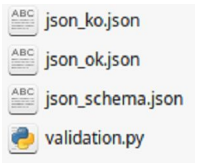
III-A. Installation▲

Pour l'installer, rien de plus simple. Le paquet, au format Wheel, est disponible sur le Pypi. Aussi, il suffit juste de faire appel à " pip ".

Sélectionnez

III-B. Utilisation avec Python▲

Voici la structure que je vous propose pour mettre en œuvre le JSON SCHEMA :



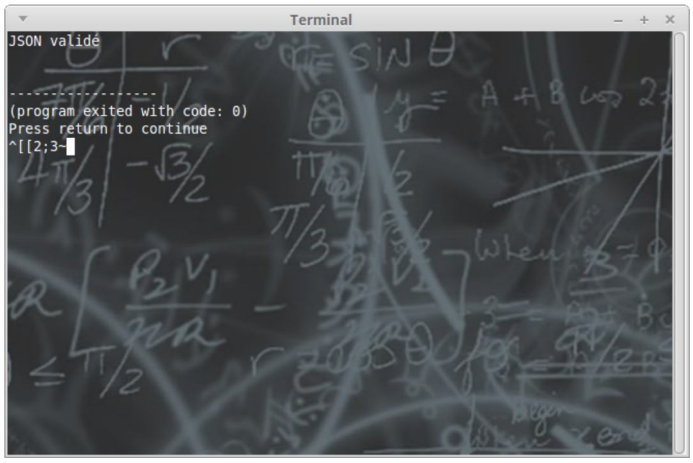
Et voici maintenant le contenu des divers fichiers. Nous allons repartir sur notre exemple initial.

Le fichier " json_ko.json " pourra simplement contenir une copie du " json_ok.json ", avec des erreurs diverses que vous pourrez éditer par vous-même afin d'expérimenter les JSON SCHEMAS.

json_ok.json
Sélectionnez
```
{
    "AGE": 12,
    "NAME": "DVP",
    "MAIL": "aa",
    "FAVORITE_COLOR": "red",
    "PHONE": "0000"
}
```
json_schema.json
Sélectionnez
```
{
    "$schema": "http://json-schema.org/schema#",
    "title": "JSON SCHEMA de demonstration",
    "description": "Ceci est un JSON SCHEMA pour le site Developpez",
    "type": "object",
    "anyOf": [{"required": ["AGE", "NAME", "PHONE"]},
              {"required": ["AGE", "NAME", "PHONE_CELL"]}],
    "properties":{"AGE": {"type": "number",
                          "minimum": 0,
                          "maximum": 130,
                          "multipleOf": 1},
                  "NAME": {"type": "string"},
                  "MAIL": {"format": "email"},
                  "FAVORITE_COLOR": {"type": "string",
                                     "enum": ["red", "blue", "yellow"]
                                    },
                  "PHONE": {"type": "string", "pattern":"^[0-9]{4}$"},
                  "CELL": {"type": "string", "pattern":"^[0-9]{4}$"}
                 }
}
```
validation.py
Sélectionnez
```
```

```
12.
13.
14.
15.
16.
17.
18.
19.
20.
21.
22.
23.
24.
import json
from jsonschema import validate

# ...

def fonction_demo(dict_to_test, dict_valid):
    try:
        validate(dict_to_test, dict_valid)
    except Exception as valid_err:
        print("Validation KO: {}".format(valid_err))
        raise valid_err
    else:
        # Realise votre travail
        print("JSON validé")


if __name__ == '__main__':
    with open("./json_ok.json", "r") as fichier:
        dict_to_test = json.load(fichier)

    with open("./json_schema.json", "r") as fichier:
        dict_valid = json.load(fichier)

    fonction_demo(dict_to_test, dict_valid)
```

L'exécution du fichier Python nous renvoie alors ce qui suit :



Comme nous pouvons le voir, rien de sorcier. La librairie jsonschema ne contient qu'une seule et unique fonction, simple d'emploi, renvoyant " True " ou bien une erreur. Dans ce dernier cas, l'élément fautif, ainsi que le type attendu, sont remontés à l'utilisateur.

Et dans le cas du test de json_ko.json, voici ce que l'écran vous renverrait (erreur sur le numéro de téléphone) :

Comme nous venons de le voir ensemble, le JSON SCHEMA vient combler un manque réel sur le format JSON : sa validation.

S'il demande un peu de temps pour être un minimum maîtrisé, cet outil est néanmoins fort pratique et parfaitement intégré à Python via la librairie adaptée : jsonschema.

Avec un développement commencé en 2010, et même si ses spécifications sont toujours en mode ' draft ', il n'en reste pas moins que l'outil est relativement mature, ou tout du moins suffisamment pour nous permettre d'améliorer nos développements.

J'espère que cet article vous aura permis d'appréhender au mieux ce concept, et vous permettra à l'avenir de mieux gérer vos interactions avec les fichiers JSON.

V. Remerciements▲

- Malick
- chrtophe
- ced