Manipulation de données

JSON et API

Xavier Gendre

Notion de document

Pas de réelle définition mais quelques propriétés :

- un **document** contient des données encodées dans un certain **format**,
- le **contenu** est un ensemble de paires **clé/valeur**,
- absence de **schéma** fixe a priori,
- un document peut contenir d'autres documents.

Exemples :

- format texte : **JSON**, XML, YML, …
- format binaire : BSON, PDF, …

Format JSON

JSON (*JavaScript Object Notation*) est un format document léger et très utilisé qui se veut lisible autant par des humains que par des machines.

La syntaxe est simple et le contenu se présente :

- comme un **objet** unique délimité par des accolades `{}`,
- ou comme une **liste d'objets** délimitée par des crochets `[]`.

Un objet contient des paires clé/valeur séparées par des virgules.

```
{
  "clé1": valeur1,
  "clé2": valeur2,
  ...
}
```

Les clés sont des chaînes de caractères délimitées par des guillemets `" "`.

Tous les caractères blancs (espace, tabulation et retour à la ligne) sont ignorés.

Une liste d’objets contient des objets séparés par des virgules.

```
[
    {...},
    {...},
    ...
]
```

Les objets sont très similaires aux dictionnaires de Python et les listes d’objets à des listes de dictionnaires.

Cette similitude permet de manipuler facilement le format JSON en Python avec le module `json` de la bibliothèque standard du langage.

Types simples

- **chaîne de caractères** : séquence de caractères Unicode délimitée par des guillemets `""`,
- **numérique** : flottants double précision (notation entière, décimale ou scientifique),
- **booléen** : `true` et `false` (en minuscules),
- **null** : valeur vide.

```
{
  "nom": "Gandalf",
  "taille": 1.68,
  "anneaux": 1,
  "magicien": true,
  "résidence": null
}
```

Types structurés

- **tableau** : liste **ordonnée** de valeurs entre crochets `[]`,
- **objet** : liste **non ordonnée** de paires clé/valeur entre accolades `{}`.

```
{
  "nom": "Gandalf",
  "stuff": ["Glamdring", "Bâton de magicien", "Narya"],
  "alias": {
    "sindarin": "Mithrandir",
    "quenya": "Olórin"
  }
}
```

Imbrication

Pour un type structuré, les valeurs peuvent être de n'importe quel type y compris un autre type structuré (*dénormalisation*).

```
{
  "nom": "Sacquet", "prénom": "Frodon",
  "amis": [
    {
      "nom": "Gamegie", "prénom": "Samsagace"
    },
    {
      "nom": "Brandebouc", "prénom": "Meriadoc"
    },
    {
      "nom": "Touque", "prénom": "Peregrin"
    }
  ]
}
```

JSON avec Python

Le module `json` de la bibliothèque standard de Python permet de manipuler des données au format JSON.

La fonction `dumps` exporte au format JSON.

```
1 import json
2
3 result = json.dumps(
4     {"nom": "Gandalf", "taille": 1.68, "residence": None}
5 )
6
7 result
```

```
'{"nom": "Gandalf", "taille": 1.68, "residence": null}'
```

```
1 type(result)
```

```
str
```

La fonction `loads` importe depuis le format JSON.

```
obj = json.loads(result)
obj
```

```
{'nom': 'Gandalf', 'taille': 1.68, 'residence': None}
```

Les fonctions `dump` et `load` sont des variantes qui permettent respectivement d'exporter au format JSON vers un fichier texte et d'importer depuis un fichier texte au format JSON.

```
with open("fichier.json", "w") as f:
    json.dump(obj, f) # Exporte en JSON

with open("fichier.json") as f:
    obj_copy = json.load(f) # Importe depuis JSON

obj_copy
```

```
{'nom': 'Gandalf', 'taille': 1.68, 'residence': None}
```

JSON avec Pandas

Pandas propose aussi des fonctions utiles pour travailler avec des données au format JSON.

La fonction `read_json` permet de créer un `DataFrame` à partir d'une chaîne de caractères contenant une list d'objets au format JSON.

```
1 import pandas as pd
2
3 df = pd.read_json(
4     '[{"nom": "Aragorn", "age": 210, "maison": "Isildur"},'
5     '{"nom": "Boromir", "age": 41, "maison": "Húrin"}]'
6 )
7 print(df)
```

```
      nom  age   maison
0  Aragorn  210  Isildur
1  Boromir   41    Húrin
```

La fonction `to_json` exporte un `DataFrame` au format JSON.

```
1 df.to_json()
```

```
'{"nom":{"0":"Aragorn","1":"Boromir"},"age":{"0":210,"1":41},"maison":
{"0":"Isildur","1":"H\\u00farin"}}'
```

Les fonctions `read_json` et `to_json` permettent aussi de manipuler des fichiers texte au format JSON.

```
1 df.to_json("fichier.json") # Exporte en JSON
2 obj_copy = pd.read_json("fichier.json") # Importe depuis JSON
3 print(obj_copy)
```

```
      nom  age   maison
0  Aragorn  210  Isildur
1  Boromir   41    Húrin
```

Il n'y a pas une unique manière de transformer des données au format JSON. L'option `orient` de `to_json` donne :

- `df.to_json(orient="columns")` : par défaut,
- `df.to_json(orient="index")` : objet unique,
- `df.to_json(orient="records")` : liste d'objets,
- `df.to_json(orient="values")` : liste de listes (perte des noms),
- `df.to_json(orient="split")` : sépare `columns`, `index` et `data`.

- Option par défaut

```
1 df.to_json(orient="columns")
```

```
'{"nom":{"0":"Aragorn","1":"Boromir"},"age":{"0":210,"1":41},"maison":
{"0":"Isildur","1":"H\\u00farin"}}'
```

- Objet unique

```
1 df.to_json(orient="index")
```

```
'{"0":{"nom":"Aragorn","age":210,"maison":"Isildur"},"1":
{"nom":"Boromir","age":41,"maison":"H\\u00farin"}}'
```

- Liste d’objets

```
1 df.to_json(orient="records")
```

```
'[{"nom":"Aragorn","age":210,"maison":"Isildur"},
{"nom":"Boromir","age":41,"maison":"H\\u00farin"}]'
```

- Liste de listes (perte des noms)

```
1 df.to_json(orient="values")
```

```
'[["Aragorn",210,"Isildur"],["Boromir",41,"H\\u00farin"]]'
```

- Sépare `columns`, `index` et `data`

```
1 df.to_json(orient="split")
```

```
'{"columns":["nom","age","maison"],"index":[0,1],"data":
[["Aragorn",210,"Isildur"],["Boromir",41,"H\\u00farin"]]}'
```

Le choix `"records"` est souvent utilisé en pratique. Grâce à l'option supplémentaire `lines=True`, il permet d'exporter au format NDJSON (*Newline Delimited JSON*).

```
print(
    df.to_json(orient="records", lines=True)
)
```

```
{"nom":"Aragorn","age":210,"maison":"Isildur"}
{"nom":"Boromir","age":41,"maison":"H\u00farin"}
```

Il faut également passer l'option `lines=True` à `read_json` pour lire un fichier au format NDJSON.

```
df.to_json("fichier.json", orient="records", lines=True) # Vers NDJ
obj_copy = pd.read_json("fichier.json", lines=True) # Depuis NDJSON
print(obj_copy)
```

```
      nom  age   maison
0  Aragorn  210  Isildur
1  Boromir   41    Húrin
```

Obtenir du JSON avec une API

De nombreux sites proposent des API (*Application Programming Interface*) pour faire des requêtes et retournent les résultats au format JSON.

Quelques exemples :

- https://lotr.fandom.com/api.php?action=opensearch&search=bombadil&limit=100
- https://lotr.fandom.com/api.php?action=query&list=allpages&format=json&aplimit=25
- https://lotr.fandom.com/api.php?action=query&titles=Hobbits&format=json&prop=extracts&explaintext&exchars=256

- Ces interfaces de programmation peuvent être utilisées dans un navigateur (peu utile), en ligne de commande (cURL, …) et, bien sûr, avec Python.
- La syntaxe d'une API donnée et le format de la réponse dépendent grandement du site web.
- La première étape avant de pouvoir utiliser une API est généralement de **lire la documentation** pour se familiariser avec les fonctionnalités.
- Chaque site peut limiter la fréquence des requêtes à son API, restreindre la taille de la réponse, imposer l'usage d'un jeton d'authentification (*token*), …

Requête simple

Pour des requêtes simples qui ne doivent retourner que des données au format JSON, la fonction `read_json` suffit 🎉

```
# Citi Bike NYC GBFS (General Bikeshare Feed Specification)
city_bike_gbfs_url = "https://gbfs.citibikenyc.com/gbfs/2.3/gbfs.js
print(
    pd.read_json(city_bike_gbfs_url)
)
```

```
                                                  data  last_updated  ttl  \
en  {'feeds': [{'url': 'https://gbfs.lyft.com/gbfs...    1746986706   60
es  {'feeds': [{'url': 'https://gbfs.lyft.com/gbfs...    1746986706   60
fr  {'feeds': [{'url': 'https://gbfs.lyft.com/gbfs...    1746986706   60

    version
en      2.3
es      2.3
fr      2.3
```

L'exemple précédent fonctionne mais le `DataFrame` doit être remis en forme pour être utilisable. Plus généralement, le JSON retourné par une API n'est toujours pas organisé d'une manière qui permette de le convertir facilement en `DataFrame` comme dans le cas du NDJSON 😞

```
lotr_fandom_url = "https://lotr.fandom.com/api.php?"
bombadil_url = (
    lotr_fandom_url + "action=opensearch&search=bombadil&limit=100"
)
print(pd.read_json(bombadil_url))
```

```
                                                   0
0                                           bombadil
1  [Tom Bombadil, Bombadil Goes Boating, Tom Bomb...
2  [, , , , , , , , , , , , , , , , , , , , , , ,...
3  [https://lotr.fandom.com/wiki/Tom_Bombadil, ht...
```

Requête avancée

Pour des requêtes plus avancées ou pour avoir accès au JSON avant de le convertir en `DataFrame`, il faut dialoguer avec un serveur web et ce n'est pas quelque chose d'évident :

- méthode de requête HTTP `GET` ou `POST`,
- gestion d'un code HTTP (200, 404, 502, …),
- données dans l'en-tête (token, …),
- …

Le module `requests` est d'une grande aide pour cela 😀

Méthode GET

La méthode HTTP `GET` permet de demander de la donnée. La fonction `get` gère très bien cette situation simple (pas d'authentification, ...).

```
1 import requests
2
3 r = requests.get(bombadil_url)
4
5 # Vérification du code HTTP
6 if r.status_code == 200:
7     print("Requête réussie, bravo !")
8 else:
9     print(f"Erreur {r.status_code}")
```

```
Requête réussie, bravo !
```

Méthode POST

La méthode HTTP `POST` est utilisée pour envoyer de la donnée au serveur afin qu'il vous réponde. Par exemple, ce sera le cas pour s'authentifier avec une clé ou un jeton.

La fonction `post` permet de traiter ces cas qui ne seront pas abordés dans la suite.

```
 1 json_url = "https://..."
 2 data = {
 3     "apikey": "MA_CLE_POUR_CETTE_API",
 4     "param": "value", ...
 5 }
 6
 7 r_post = requests.post(json_url, data=data)
 8
 9 # Vérification du code HTTP
10 if r_post.status_code == 200:
11     ...
```

Réponse

L'objet retourné par les fonctions `get` ou `post` contient toute l'information de la requête dont la réponse JSON du serveur.

- `r.text` : chaîne de caractères au format JSON (avec encodage)

```
1 r.text
```

```
'["bombadil",["Tom Bombadil","Bombadil Goes Boating","Tom Bombadil\'s so...'
```

- `r.json()` : importe depuis la réponse JSON

```
1 r.json()
```

```
['bombadil',
 ['Tom Bombadil',
  'Bombadil Goes Boating',
  "Tom Bombadil's songs",
```

```
'Theories about Tom Bombadil',
'The Adventures of Tom Bombadil and Other Verses from the Red Book',
'The Adventures of Tom Bombadil',
'The Adventures of Tom Bombadil (disambiguation)',
'In the House of Tom Bombadil',
'Preface to The Adventures of Tom Bombadil',
'Sauron',
'Gandalf',
'Aragorn II',
'The Lord of the Rings',
'Frodo Baggins',
'Middle-earth'
```

L’objet importé peut maintenant être manipulé pour créer un `DataFrame` adapté aux besoins.

```
obj = r.json()
data = {
    "page_nom": obj[1],
    "page_url": obj[3],
}

print(pd.DataFrame(data).head(3))
```

```
                page_nom                                           page_url
0           Tom Bombadil          https://lotr.fandom.com/wiki/Tom_Bombadil
1  Bombadil Goes Boating  https://lotr.fandom.com/wiki/Bombadil_Goes_Boa...
2   Tom Bombadil's songs  https://lotr.fandom.com/wiki/Tom_Bombadil%27s_...
```

À vous de jouer !