TP - Construire un corpus XML à partir du web

Alexander Delaporte

Construire un corpus XML à partir du web

Dans cet exercice, vous allez produire un corpus XML très simple à partir de données extraites d'une page web. Les étapes à suivre sont les suivantes :

1. Déclarer l'arborescence XML.
2. Extraire les données d'une page web pour peupler l'arborescence.
3. Enrichir les données XML.
4. Enregistrer le corpus XML dans un fichier.

Les données que vous allez traiter sont issues d'un petit lexique thématique français-japonais, constitué d'informations que j'ai collectées moi-même, manuellement, dans des dictionnaires bilingues japonais-français. Elles sont rassemblées dans un fichier TSV (il s'agit d'un format CSV dont le caractère séparateur est une tabulation plutôt qu'une virgule), qui est mis à disposition sur Github.

L'interface de Github vous permet de consulter le fichier sous forme de tableau, dans lequel vous pouvez effectuer des recherches à partir de chaînes de caractères. Vous pouvez conserver cette vue dans un onglet pour garder un aperçu du tableau pendant votre travail, mais l'URL que nous allons interroger dans le script ne sera pas celle de cette page. Ce sera celle de la visualisation dite "raw", c'est-à-dire qui présente le contenu du fichier seul, sans aucune forme d'interface graphique.

En résumé :

- URL à consulter pour avoir un aperçu des données à traiter = https://github.com/alxdrdelaporte/TEKIPAKI/blob/master/lexique_linguistique_japonais.tsv
- URL à utiliser dans le script pour extraire les données = https://github.com/alxdrdelaporte/TEKIPAKI/raw/master/lexique_linguistique_japonais.tsv

C'est parti ! 🚀

Déclarer l'arborescence XML

Nous allons créer un petit corpus au format XML à partir de ces données tabulaires. Dans le tableau, chaque ligne correspond à un item, représenté par ses différentes formes :

- 3 formes pour la version japonaise, avec l'écriture en kanji (quand elle existe), la transcription en hiragana ou katakana, et la transcription en caractères latins, dits *rōmaji*.
- L'équivalent traductionnel en français.

En termes de structure XML, nous allons donc simplement créer un élément `<item>` pour chaque ligne du tableau, qui contiendra 4 sous-éléments correspondant à ses 4 formes écrites : `<kanji>`, `<hiragana_katakana>`, `<romaji>`, `<francais>`. Pour faciliter d'éventuels traitement automatiques ultérieurs, nous ajouterons également un attribut `id` à chaque élément `<item>`.

```
<LEXIQUE>
    <item id="1">
        <kanji>...</kanji>
        <hiragana_katakana>...</hiragana_katakana>
```

```
        <romaji>...</romaji>
        <francais>...</francais>
    </item>
    <!-- (...) -->
</LEXIQUE>
```

Dans le script, la création de l'arborescence XML commence par la déclaration de sa racine, ici `<LEXIQUE>`.

Commençons par importer la librairie `xml.etree.ElementTree` qui permettra à Python de gérer le XML. Exécutez la cellule ci-dessous sans la modifier, soit en cliquant sur le bouton `Exécuter`, soit en utilisant les touches `shift` + `enter` ou `ctrl` + `enter`.

In [ ]:
```
import xml.etree.ElementTree as ET
```

Les fonctions de la librairie `xml.etree.ElementTree` pourront désormais être appelées en utilisant la syntaxe `ET.nom_de_la_fonction()`. Sans `as ET` dans la ligne d'import, la syntaxe à utiliser serait `xml.etree.ElementTree.nom_de_la_fonction()`.

La racine de la structure XML est un élément qui n'a pas besoin d'être rattaché à un élément parent. Ce cas est à représenter par une instance d'objet de la classe `Element`, issue de `ET`, en utilisant la syntaxe `ET.Element("Nom de l'élément")`. Dans la cellule ci-dessous, déclarez la racine du corpus et associez-la à la variable `racine`.

In [ ]:
```
racine = # Déclarez la racine du corpus
```

Voici 2 fonctions qui vous aideront pour la suite :

- La fonction `xml2str()` convertit le XML en chaîne de caractères. Vous n'aurez pas besoin d'appeler directement cette fonction, elle sera appelée par d'autres fonctions.
- La fonction `afficher_xml()` permet d'afficher une arborescence XML. Elle prend en paramètre la variable correspondant à la racine de l'arborescence.

Vous n'avez pas besoin de modifier le contenu de la cellule ci-dessous, mais n'oubliez pas de l'exécuter avant de passer à la suite pour pouvoir utiliser ces deux fonctions.

In [ ]:
```
def xml2str(racine_xml):
    return ET.tostring(racine_xml, encoding="unicode", method="xml")


def afficher_xml(racine_xml):
    print(xml2str(racine_xml))
```

La fonction `afficher_xml()` étant maintenant disponible, utilisez-la pour afficher la structure contenue dans `racine`.

In [ ]:
```
# Affichez la structure XML à l'aide de la fonction fournie
```

Extraire les données de la page web pour peupler l'arborescence

Pour récupérer les données d'une page web, il faut envoyer une requête à son URL, ce que permet de faire la librairie `requests`. Exécutez la cellule ci-dessous pour l'importer.

In [ ]:
```
import requests
```

Envoyer une requête à l'URL cible

Ce n'est pas forcément obligatoire, mais comme les URL peuvent être des chaînes de caractère relativement longues, il est plus confortable d'associer l'URL à une variable. Associez à la variable `url` l'URL que nous voulons interroger ici.

In [ ]:
```
url = # Associez l'URL appropriée à la variable url
```

C'est la fonction `.get()` qui va soumettre la requête ; puisqu'elle est issue de `requests`, la syntaxe à utiliser est `requests.get(url_à_interroger)`. Utilisez cette fonction sur l'URL de la page qui contient les données à extraire, et associez son résultat à la variable `page`.

In [ ]:
```
page = # Interrogez la page web
```

Nous pouvons maintenant consulter le résultat de la requête en appelant la variable `page`. Appelez `page` dans la cellule ci-dessous.

In [ ]:
```
# Appelez la variable page
```

S'il n'y a pas de problème, vous obtenez `<Response [200]>`.

Récupérer le code source

200 est le code de réponse HTTP obtenu, et indique le succès de la requête. Le code source de la page interrogée est accessible via l'attribut `text` de `page` (`page.text`). Appelez cet attribut, en l'associant à la variable `donnees`.

In [ ]:
```
donnees = # Associez à donnees l'attribut text de page
```

Appelez `donnees` pour visualiser son contenu :

In [ ]:
```
# Appelez la variable donnees
```

Vous obtenez normalement quelque chose qui ressemble à une assez longue chaîne de caractères qui commence comme ceci :

```
'Kanji\tHiragana/Katakana\tRōmaji (Hepburn)\tFrançais\n付録\tふろく
\tfuroku\tappendice\n辞典\tじてん\tjiten\tdictionnaire, lexique\n英和辞典\tえいわ
じてん\teiwa jiten\tdictionnaire anglais-japonais\n英英辞典\tえいえいじてん\teiei
jiten\tdictionnaire anglais-anglais\n仏和辞典\tふつわじてん\tfutsuwa
jiten\tdictionnaire français-japonais\n
(...)
```

Traiter le code source pour déclarer et alimenter les sous-éléments XML

Est-ce vraiment une chaîne de caractères ? Vérifions avec la fonction `type()`, qui renvoie le type de données de ce qui lui est passé en paramètre.

In [ ]:
```
type(donnees)
```

Comme `type(donnees)` renvoie `str` (si ce n'est pas le cas, il y a une erreur dans les étapes précédentes !), notre variable `donnees` contient bien une chaîne de caractères. Il est alors possible de lui appliquer les méthodes et fonctions propres à ce type de données.

Nous avons donc maintenant le contenu de la page web stocké dans la variable `donnees`. S'agissant d'une page dépourvue de formattage HTML, aucun balisage HTML n'a été récupéré, il n'y a pas lieu de s'en soucier : `donnees` contient uniquement les informations à récupérer pour alimenter l'arborescence XML.

Pour rappel, le but est ici d'obtenir un corpus XML dans lequel chaque ligne du tableau de données est représentée par un élément `<item>`. Segmenter le contenu de `donnees` selon les lignes constitue un bon point de départ pour la construction de la structure XML. Les chaînes de caractères (= type `str`) disposent d'une méthode `split()`, qui s'applique avec la syntaxe `chaine_de_caracteres.split(separateur)`.

Utilisez `split()` pour séparer les lignes du texte stocké dans `donnees`. Associez le résultat à la variable `lignes`.

In [ ]:
```
lignes = # Segmentez le texte contenu dans donnees selon les lignes
```

Appelez `lignes` pour visualiser son contenu.

In [ ]:
```
# Appelez la variable lignes
```

S'il n'y a pas d'erreur, `lignes` contient maintenant une collection de chaînes de caractères, correspondant chacune à une ligne du tableau. Plus précisément, il s'agit d'une *liste* (type `list`). Comme l'indique la documentation, celle-ci peut être parcourue avec une boucle `for`.

C'est ce que nous allons utiliser pour créer un élément `<item>` pour chaque ligne. Contrairement à l'élément racine `<LEXIQUE>` déclaré précédemment, chacun de ces éléments `<item>` est en fait un sous-élément rattaché à l'élément parent `<LEXIQUE>`. Leur déclaration ne se fera donc pas avec `ET.Element("Nom de l'élément")`, mais cette fois avec `ET.SubElement(element_parent, "Nom de l'élément")`.

Complétez la boucle `for` ci-dessous pour déclarer, pour chaque ligne, un élément `<item>` rattaché à la racine `<LEXIQUE>`.

In [ ]:
```
item_id = 1

for ligne in lignes:
    item = # Créez un sous-élément "item"
```

Affichez la structure XML pour vérifier qu'il n'y a pas de problème. Elle devrait se composer de la racine `<LEXIQUE>`, contenant une suite de sous-éléments `<item>` vides.

In [ ]:
```
# Affichez l'arborescence XML
```

Avant d'attribuer un contenu à ces éléments `<item>`, il faut leur ajouter leur identifiant. Dans la cellule ci-dessous, reprenez le code de la boucle `for` et complétez-le pour ajouter à la boucle la déclaration d'un attribut `id`, de valeur croissante (associée à une variable nommée `item_id`), pour chaque élément `<item>`.

Quelques indices pour accomplir cette tâche :

- La syntaxe pour ajouter un attribut à un élément XML est la suivante : `element.set("Nom de l'attribut", valeur)`.
- Pour incrémenter de 1 une valeur, vous pouver utiliser `valeur = valeur + 1` ou `valeur += 1`.
- Seuls les types numériques peuvent être incrémentés, mais il est préférable que la valeur de l'attribut `id` soit une chaîne de caractères (`str`). Pour convertir un objet en chaîne de caractères, la fonction à utiliser est `str(objet)`.

In [ ]:
```
racine = ET.Element("LEXIQUE")
item_id = 1

for ligne in lignes:
    item = # Créez un sous-élément "item"
    # Ajoutez l'attribut "id"
    # Incrémentez la variable item_id
```

Affichez la structure XML pour vous assurer que chaque `<item>` a un attribut `id` et que la valeur de cet attribut est cohérente.

In [ ]:
```
# Affichez la structure XML
```

La structure des éléments `<item>` est prête, il est temps d'ajouter leur contenu. Dans l'arborescence à mettre en place, chaque `<item>` contient 4 sous-éléments, correspondant aux 4 formes écrites de l'item. Dans le tableau de données, ces 4 formes sont représentées par les 4 cellules d'une même ligne.

La segmentation par ligne est déjà effectuée et parcourue par la boucle `for`, il reste donc à diviser la chaîne de caractères `ligne` pour isoler le contenu correspondant à chaque cellule.

Complétez le code ci-dessous pour :

1. Segmenter `ligne` suivant ses cellules, en stockant la liste obtenue dans la variable `segments`.
2. Attribuer la valeur de la cellule *Kanji* à la variable `segment_kanji`, celle de la cellule *Hiragana/Katakana* à la variable `segment_hiragana_katakana`, etc. Pour ce faire, sachez que vous pouvez appliquer un index à une liste pour en extraire un élément donné.

Note : En Python, les index débutent à 0 et non à 1.

In [ ]:
```
racine = ET.Element("LEXIQUE")
item_id = 1

for ligne in lignes:
    """Copiez-collez votre code (début)"""
    item = # Créez un sous-élément "item"
    # Ajoutez l'attribut "id"
    # Incrémentez la variable item_id
    """Copiez-collez votre code (fin)"""
    segments = # Segmentez la ligne selon les cellules
    # Assignez des variables aux segments
    segment_kanji =
    segment_hiragana_katakana =
    segment_romaji =
    segment_francais =
    # Vérifiez que tout est OK
    print(segment_kanji, segment_hiragana_katakana, segment_romaji, segment_francais)
```

Et voilà, c'est parfait ! ... À deux détails près :

1. La boucle a récupéré la ligne d'étiquettes "Kanji Hiragana/Katakana Rōmaji (Hepburn) Français". C'est normal, mais nous n'en voulons pas dans le corpus XML !
2. Exécuter le code de la cellule ci-dessus lève une erreur `IndexError: list index out of range`.

Laissons de côté le message d'erreur pour le moment. Pour ne pas récupérer la ligne d'étiquettes, il s'agit en fait de passer la première ligne du tableau sans la traiter : vous savez déjà quasiment comment faire.

Complétez la condition `if` dans la cellule ci-dessous pour ignorer la première ligne :

In [ ]:
```
racine = ET.Element("LEXIQUE")
item_id = 1

for ligne in lignes:
    # Ignorez la ligne d'étiquettes
    if ligne is not XXX:
        """Copiez-collez votre code (début), attention à l'indentation"""
        item = # Créez un sous-élément "item"
        # Ajoutez l'attribut "id"
        # Incrémentez la variable item_id
        segments = # Segmentez la ligne selon les cellules
        # Assignez des variables aux segments
        segment_kanji =
        segment_hiragana_katakana =
        segment_romaji =
        segment_francais =
        # Vérifiez que tout est OK
        print(segment_kanji, segment_hiragana_katakana, segment_romaji, segment_francais
        """Copiez-collez votre code (fin)"""
```

Intéressons-nous maintenant à l'`IndexError` qui, en toute logique, est encore présente. Regardez le message d'erreur complet : il nous indique que la description de l'erreur est `list index out of range` et qu'elle est survenue lors de l'assignation d'une valeur à la variable `segment_hiragana_katakana` (signalé par la flèche et le numéro de ligne en caractères gras).

Ceci signifie que, lors du traitement d'une ligne du tableau, il n'a pas été possible d'atteindre l'index correspondant à la cellule *Hiragana/Katakana* : en d'autres termes, le script s'est heurté à une ligne comportant une unique cellule !

Avez-vous compris ce qu'il s'est passé ? Si ce n'est pas (encore) le cas, observez attentivement le tableau de données, l'arborescence XML actuellement contenue dans `racine`, ainsi que le contenu des variables `donnees` et `lignes`, tout en gardant en tête que la ligne d'étiquettes a été ignorée **et** que les index en Python débutent à 0.

Une fois le problème identifié, vous savez normalement comment le résoudre.

L'ajout de l'opérateur `and` permet d'ajouter une nouvelle condition au `if` déjà mis en place. Complétez la condition `if` pour enfin vous débarrasser de l'erreur `IndexError`.

In [ ]:
```
racine = ET.Element("LEXIQUE")
item_id = 1

for ligne in lignes:
    # Ignorez la ligne d'étiquettes + évitez de lever IndexError
    if ligne is not XXX and XXX:
        """Copiez-collez votre code (début), attention à l'indentation"""
        item = # Créez un sous-élément "item"
        # Ajoutez l'attribut "id"
        # Incrémentez la variable item_id
```

```
        segments = # Segmentez la ligne selon les cellules
        # Assignez des variables aux segments
        segment_kanji =
        segment_hiragana_katakana =
        segment_romaji =
        segment_francais =
        # Vérifiez que tout est OK
        print(segment_kanji, segment_hiragana_katakana, segment_romaji, segment_francais
        """Copiez-collez votre code (fin)"""
```

Maintenant qu'il n'y a plus de problème, il ne vous reste plus qu'à ajouter ces données à l'arborescence XML. Créez d'abord des sous-éléments :

In [ ]:
```
racine = ET.Element("LEXIQUE")
item_id = 1

for ligne in lignes:
    """Copiez-collez votre code (début)"""
    # Ignorez la ligne d'étiquettes + évitez de lever IndexError
    if ligne is not XXX and XXX:
        item = # Créez un sous-élément "item"
        # Ajoutez l'attribut "id"
        # Incrémentez la variable item_id
        segments = # Segmentez la ligne selon les cellules
        # Assignez des variables aux segments
        segment_kanji =
        segment_hiragana_katakana =
        segment_romaji =
        segment_francais =
        """Copiez-collez votre code (fin)"""
        # Créez des sous-éléments "kanji", "hiragana_katakana", "romaji" et "francais"
```

Vérifiez que les sous-éléments ont bien été créés :

In [ ]:
```
# Vérifiez la création des sous-éléments
```

En l'état, les sous-elements sont maintenant présents dans la structure XML mais dépourvus de contenu. Le contenu textuel de chaque (sous-)élément XML se trouve dans son attribut `text` (`element.text`).

Ajoutez le contenu textuel correspondant aux différents sous-éléments de chaque `item` de notre corpus.

In [ ]:
```
racine = ET.Element("LEXIQUE")
item_id = 1

for ligne in lignes:
    """Copiez-collez votre code (début)"""
    # Ignorez la ligne d'étiquettes + évitez de lever IndexError
    if ligne is not XXX and XXX:
        item = # Créez un sous-élément "item"
        # Ajoutez l'attribut "id"
        # Incrémentez la variable item_id
        segments = # Segmentez la ligne selon les cellules
        # Assignez des variables aux segments
        segment_kanji =
        segment_hiragana_katakana =
        segment_romaji =
        segment_francais =
        # Créez des sous-éléments "kanji", "hiragana_katakana", "romaji" et "francais"
        """Copiez-collez votre code (fin)"""
        # Ajoutez le contenu textuel des sous-éléments
```

Affichez le corpus pour vérifier qu'il n'y a pas de problème.

In [ ]:
```
# Affichez le corpus
```

Maintenant qu'il y a une certaine quantité de contenu dans l'aborescence, l'afficher de cette façon manque de lisibilité. Exécutez la cellule ci-dessous pour obtenir deux nouvelles fonctions qui vous permettront d'obtenir une visualisation plus agréable pour un oeil humain.

In [ ]:
```
from bs4 import BeautifulSoup as soup


def joli_xml(racine_xml):
    # Le parseur "xml" permet l'ajout automatique de la déclaration XML
    # Si vous rencontrez un message d'erreur lié au parseur, passez à "html.parser" comm
    # return soup(xml2str(racine_xml), "html.parser").prettify()
    return soup(xml2str(racine_xml), "xml").prettify()


def afficher_joli_xml(racine_xml):
    print(joli_xml(racine_xml))
```

Vous n'avez pas besoin d'utiliser directement `joli_xml()` pour le moment. Visualisez le corpus avec `afficher_joli_xml()`.

In [ ]:
```
# Utiliser afficher_joli_xml() pour visualiser le corpus
```

Enrichir les données XML

Le corpus a maintenant été alimenté avec toutes les données du tableau. Avant de l'enregistrer dans un fichier, nous allons lui ajouter quelques informations supplémentaires.

Ajouter un attribut

Il se trouve que les sources dont sont issus les items de ce lexique sont connues, mais elles ne figurent pas dans le tableau. Nous allons créer un nouvel attribut `source` pour chaque élément `item` du corpus afin d'y faire figurer cette information.

```
<LEXIQUE>
    <item id="1" source="...">
        <japonais>
            <kanji>...</kanji>
            <hiragana_katakana>...</hiragana_katakana>
            <romaji>...</romaji>
        </japonais>
        <francais>...</francais>
        <commentaire>...</commentaire>
    </item>
    <!-- Etc. -->
</LEXIQUE>
```

Les sources sont les suivantes.

De 付録-ふろく-*furoku-appendice* à 常-つね-*tsune-normal, habituel* :

- KURAKATA et al. (2003) プチ・ロワイヤル仏和辞典 (*Nouveau Petit Royal dictionnaire français -japonais*), troisième édition. Tokyo : Obunsha.
- TSUNEKAWA et al. (2002) *Petit dictionnaire japonais – français Royal*. Tokyo : Obunsha.

De 副詞-ふくし-*fukushi-adverbe* à 最上級-さいじょうきゅう-*saijoukyuu-superlatif* :

- *Sanseido's New Concise English-Japanese Dictionary, Revised Edition* | 新コンサイス英和辞典・第2版 (1985). Tokyo : Sanseido.

La première partie du lexique est issue de deux dictionnaires bilingues français-japonais édités par Obunsha, la seconde d'un dictionnaire bilingue anglais-japonais édité par Sanseido. Nous allons utiliser le nom de l'éditeur comme source, l'attribut `source` aura donc pour valeur soit "Obunsha", soit "Sanseido".

Mais comment procéder ? Nous avons construit et stocké dans la variable `racine` une structure XML. Tout comme il existe des fonctions permettant d'écrire une arborescence XML, il y a des fonctions qui permettent de la lire. `find()` et `findall()` sont des méthodes propres aux objets `xml.etree.ElementTree.Element` qui permettent de retrouver un élément à partir de son nom.

- `find()` s'utilise avec la syntaxe `element_parent.find("Nom de l'élément recherché")` et renvoie uniquement le premier élément correspondant.
- `findall()` s'utilise avec la syntaxe `element_parent.findall("Nom de l'élément recherché")` et renvoie une liste de l'ensemble des éléments correspondants.

Récupérez la liste des éléments `item` du corpus et associez-la à la variable `items`.

In [ ]:
```
items = # Récupérez la liste des éléments item
```

Cette liste peut être parcourue avec une boucle `for`, mais pour attribuer la valeur du futur attribut `source` il faudrait plutôt pouvoir parcourir séparément les deux parties de la liste correspondant aux items issus de chaque source. C'est bien sûr possible, mais il faut au préalable connaître les index des éléments `item` (dans la liste `items`) qui se situent à la limite entre les deux sources.

Il s'agit de 常-つね-*tsune-normal, habituel*, dernier item issu des dictionnaires Obunsha, et 副詞-ふくし-*fukushi-adverbe*, premier item issu du dictionnaire Sanseido.

Trouvez l'index de 常-つね-*tsune-normal, habituel* dans la liste `items` et, dans la cellule ci-dessous, vérifiez qu'il s'agit du bon index selon la méthode de votre choix.

In [ ]:
```
# Vérifiez l'index de 常-つね-tsune-normal, habituel
```

Nous allons utiliser une boucle `for` pour parcourir la partie de la liste `items` qui correspond aux données issues des dictionnaires Obunsha. Le nombre d'itérations à effectuer peut être réglé grâce à la fonction `range()`.

En vous aidant de la documentation de la fonction `range()` et en réutilisant ce que vous avez écrit dans la cellule précédente, complétez la boucle `for` pour vérifier que vous avez sélectionné la bonne portion de la liste `items` (c'est-à-dire de 付録-ふろく-*furoku-appendice* à 常-つね-*tsune-normal, habituel*).

In [ ]:
```
for i in range(XXX):
    # Vérifiez la portion de la liste items parcourue par la boucle
```

La déclaration d'un attribut pour un élément de l'arborescence XML se fait en utilisant la syntaxe

`element.attrib["Nom de l'attribut"] = "Valeur"`.

Si votre boucle `for` parcourt correctement les éléments allant de 付録-ふろく-*furoku-appendice* à 常-つね-*tsune-normal, habituel*, modifiez la cellule précédente ou utilisez la cellule suivante pour leur ajouter un attribut `source` dont la valeur est "Obunsha".

In [ ]:
```
for i in range(XXX):
    # Ajoutez un attribut source dont la valeur est "Obunsha"
```

Faites la même chose pour les items issus du dictionnaire édité par Sanseido, allant de 副詞-ふくし-*fukushi-adverbe* à 最上級-さいじょうきゅう-*saijoukyuu-superlatif*.

Vous aurez certainement besoin de la fonction `len(liste)`, qui renvoie la longueur d'un objet de type `list` (= le nombre d'éléments qui composent la liste).

In [ ]:
```
# Vérifiez la portion de la liste parcourue par la boucle
```

In [ ]:
```
# Ajoutez un attribut source dont la valeur est "Sanseido"
```

In [ ]:
```
# Vérifiez que les attributs ont été correctement ajoutés
```

Créer un (sous-)élément supplémentaire

Vous avez peut-être remarqué en observant le tableau contenant le lexique que pour certaines lignes, la contenu de la cellule *Kanji* et celui de la cellule *Hiragana/Katakana* sont identiques. Il s'agit des lignes correspondant, en français, aux termes *liaison* et *élision*. Ce n'est pas dû à une erreur ou une donnée manquante : la liaison et l'élision étant des phénomène du français qui n'existent pas en japonais, leur traduction est en fait une simple transcription en katakana.

Nous allons ajouter aux éléments `<item>` correspondant à *liaison* et *élision* un sous-élément `<commentaire>Pas d'écriture en kanji.</commentaire>`. En fait, vous savez déjà comment faire !

Commencez par associer aux variables `liaison` et `élision` les items correspondants (dans la liste `items`). Sachant que `find()` et `findall()` peuvent aussi prendre des expressions Xpath en paramètre, vous pouvez vous amuser à trouver différentes façons d'y parvenir.

In [ ]:
```
liaison = # Associez l'item correspondant
elision = # Associez l'item correspondant

# Utilisez print() pour vérifier le contenu des variables liaison et elision
```

Créez maintenant pour ces deux éléments un sous-élément `<commentaire>` dont le contenu textuel est, par exemple, "Pas d'écriture en kanji.". Vous pouvez le faire avec une boucle `for`, ou sans boucle. Vous pouvez écrire une fonction si vous le souhaitez.

In [ ]:
```
# Pour les items liaison et elision, créez un sous-élément <commentaire> et son contenu
```

In [ ]:
```
# Vérifiez que les sous-éléments <commentaire> ont bien été créés
```

Vous avez réussi ? Bravo, il ne reste plus qu'à enregistrer votre corpus dans un fichier !

Enregistrer le corpus XML dans un fichier

Effectivement, même si vous avez maintenant obtenu un corpus au format XML conforme à vos objectifs, le seul endroit où il est sauvegardé est la variable `racine`. Comme vous l'avez vu lors de l'étape d'enrichissement des données, ça ne pose aucun problème pour lui appliquer différents traitements.

Cependant, il peut bien entendu être nécessaire de produire un fichier XML, pour diverses raisons : partage et diffusion du corpus, confort de travail, changement d'outil de traitement... Ou tout simplement parce que ce corpus est le résultat final que vous cherchiez à obtenir.

Je vous fournis dans la cellule ci-dessous la syntaxe de base pour effectuer l'écriture du corpus dans un fichier de sortie au format XML. L'ouverture du fichier se fait avec la fonction `open()`, ses paramètres sont :

- Le chemin d'accès du fichier de sortie, ici `"lexique.xml"`.
- Le mode d'ouverture du fichier. Nous voulons ici écrire un fichier, il faut donc utiliser `"w"` (*write*). Les autres valeurs possibles sont `"r"` (*read*), `"a"` (*append*) et `"x"` (*create*).
- L'encodage des caractères, ici `"utf-8"`.

L'écriture du contenu dans le fichier ouvert se fait via la méthode `write()`. Complétez la cellule suivante de façon à enregistrer votre corpus dans le fichier *lexique.xml*.

In [ ]:
```
with open("lexique.xml", "w", encoding = "utf-8") as lexique:
    lexique.write(XXX) # Enregistrez votre corpus XML dans le fichier lexique.xml
```

Ouvrez le fichier pour vérifier son contenu :

- Le fichier contient-il quelque chose ?
- Tous les caractères s'affichent-ils correctement ?
- Le XML est-il bien formé ?
- L'arborescence est-elle conforme à la structure attendue ?

Si votre réponse est *oui* pour l'ensemble de ces questions, félicitations !

🏆 Vous avez construit un corpus XML à partir d'une page web.