Piles et Files

Terminale NSI - Travaux dirigés

qkzk

2020/01/01

Table des matières

Motivation

Fichier au format HTML bien formés

Le HTML est un format de fichier utilisé par les navigateurs web. Les fichiers au format HTML (et plus généralement au format XML) sont des fichiers texte dans lesquels on trouve des balises :

- ouvrantes de la forme `<nom attributs>`,
- fermantes de la forme `</nom>`,
- auto-fermantes de la forme `<nom attributs/>`.

où `nom` désigne le nom de la balise et `attributs` une liste de couples `clé=valeur`.

Dans la pratique, `nom` est par exemple `div`, `p`, `html`, `body`, `head`, ... Pour la syntaxe des tags HTML, voir ici. Les balises auto-fermantes sont : `area`, `br`, `hr` , `img` , `input` , `link` , `meta` et `param`.

Dans cette activité, on considère qu'un document HTML est bien formé si :

- à chaque balise ouvrante correspond une balise fermante,
- on ne peut fermer une balise que si toutes les balises situées entre les deux balises ouvrantes et fermantes sont fermées.

Par exemple, les documents ex1.html et ex4.html sont bien formés, alors que les documents ex2.html et ex3.html sont mal formés.

On veut écrire un prédicat renvoyant `True` si un texte est un document HTML bien formé et `False` dans le cas contraire.

Dans ce cadre, on peut donc ne pas tenir compte des balises auto-fermantes (et on ne demande pas de vérifier ici si les balises sont des balises HTML existantes, ni si on respecte les attributs des balises).

La première étape est d'écrire un parser de fichier HTML, permettant de parcourir séquentiellement les balises.

Écriture d'un parser HTML

L'écriture d'un tel parser est une tâche difficile, car un caractère `<` peut être rencontré dans différentes situations :

- Situation 1 : définition du type du document : `<!DOCTYPE ...>`
- Situation 2 : signe inférieur dans le texte : `i < len(l)`
- Situation 3 : commentaires HTML `<!-- -->`
- Situation 4 : signe inférieur dans des attributs : `<script data-user=">myfunc();">`

Pour simplifier les choses nous commencerons par traiter uniquement des fichiers HTML avec la situation 1, puis nous autoriserons dans une deuxième version les documents HTML contenant les situations 2 à 4.

Structure de file

Les tags rencontrés seront ajoutés dans une file. Une file est une structure de données séquentielle agissant comme une file d'attente : on peut ajouter et retirer des éléments de la structure, mais les premiers éléments ajoutés seront toujours les premiers à sortir.

La structure de file dispose des primitives suivantes :

- `q=Queue()` : crée une nouvelle file vide ;
- `q.enqueue(el)` : ajoute l'élément `el` à la file `q` ;
- `q.dequeue()` : enlève le premier élément ajouté à la structure et le renvoie ;
- `q.is_empty()` : renvoie `True` si, et seulement si, la file est vide.

On dit que la file est une structure **FIFO** (*First In First Out*).

Le fichier queue_squel.py contient le squelette d'un module implémentant une structure de file. * que proposez-vous pour représenter une file ? * renommez de fichier squelette en `myqueue.py` et complétez le (on ne peut utiliser le nom `queue.py` car Python dispose déjà d'un module homonyme).

```
class QueueEmptyError(exception):
    def __init__(msg):
        super().__init__(msg)

class Queue:
    """
    a class representing a queue

    >>> q = Queue()
    >>> q.is_empty()
    True
    >>> q.enqueue(1)
    >>> q.enqueue(2)
    >>> q.is_empty()
    False
    >>> q.dequeue()
    1
    >>> q.dequeue()
    2
    """

    def __init__(self):
        """
        create a new queue
        """
        pass

    def is_empty(self):
        """
```

```
        :return: (bool) True si la queue est vide, False sinon
        """
        pass

    def enqueue(self, el):
        """
        enfile un élément dans la file
        :param el: (any) un élément
        """
        pass

    def dequeue(self):
        """
        défile un élément
        :return: (any) un élément
        """
        pass

if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose = True)
```

On rappelle que les listes offrent les fonctionnalités ci-dessous et on peut noter que la documentation Python précise : "*Alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertions ou de retraits en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).*" :

```
>>> l= [1,2,3,4]
>>> l
[1, 2, 3, 4]
>>> l[0]
1
>>> l[-1]
4
>>> l[1:]
[2, 3, 4]
>>> l.append(5)
>>> l
[1, 2, 3, 4, 5]
>>> l.pop()
5
>>> l
[1, 2, 3, 4]
```

Et aussi, éventuellement :

```
>>> del l[0]
>>> l
[2, 3, 4]
```

Première version

Dans un module `html_parser1.py`, écrire une première version du parser : la fonction `parse(document)` prend en paramètre un document sous forme d'une chaîne de caractères et renvoie une file contenant les tags.

Pour vous aider, le fichier html_parser_squel.py contient le squelette d'un parser.

Dans cette première version, on considère que seule la situation 1 peut être rencontrée et on ignore les situations 2, 3 et 4. Et on rappelle que les balises auto-fermantes peuvent être ignorées.

Par souci de simplification on pourra considérer que les attributs d'une balise ne peuvent être séparés que par des espaces.

On utilisera à profit la méthode `index` des chaînes de caractères.

```
Help on method_descriptor:

index(...)
    S.index(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Raises ValueError when the substring is not found.
```

La méthode `split` (documentation) peut également s'avérer utile.

Vérificateur HTML : un algorithme

Dès que l'on peut récupérer les tags séquentiellement, nous pouvons nous attaquer à l'écriture du vérificateur HTML (checker). Il peut être utile de faire l'analogie entre les documents HTML et les expressions correctement parenthésées.

Une expression bien parenthésée est une expression contenant un certains nombre de type de parenthèses telle que :

- à chaque parenthèse ouvrante correspond une parenthèse fermante,
- à tout moment, on ne peut fermer une parenthèse que si l'expression située entre les deux parenthèses se correspondant est bien parenthésée.

Par exemple, les expressions suivantes sont bien parenthésées:

- `((()))`
- `()()()`
- `([{}]())`

Les expressions suivantes ne sont pas bien parenthésées :

- `(()))`
- `(()`
- `([)]`

Pour vérifier si une expression est correctement parenthésée, une méthode est d'utiliser une pile :

- lorsque l'on rencontre une parenthèse ouvrante on l'empile,
- lorsque l'on rencontre une parenthèse fermante, par exemple `]` :
  - si la pile est vide, alors l'expression est mal parenthèsée (trop de fermantes),
  - sinon, on dépile l'ouvrante située au sommet de la pile et on vérifie que les deux parenthèses correspondent (`(` et `)`, `[` et `]`).

Lorsque toute la chaîne a été parcourue, la pile doit être vide (pas d'ouvrante non fermée).

Un document HTML peut être vue comme une expression parenthésée :

- les tags ouvrants `<tag>` sont les parenthèses ouvrantes;
- les tags fermants `</tag>` sont les parenthèses fermantes.

Sur la pile, on placera des tags ouvrants.

Opération primitives sur les piles

Les *piles* sont des structures de données linéaires admettant les *opérations primitives* suivantes :

- création d'une pile vide
- empilement d'un élément sur une pile
- dépilement du sommet d'une pile
- test de vacuité d'une pile.

Le principe d'une pile est le suivant : on ne peut accéder un élément (e) de la pile qu'en ayant enlevé d'abord tous les éléments empilés après (e).

L'élément renvoyé par la méthode `pop` est le dernier élément empilé. On dit que la pile est une structure **LIFO** (*Last In First Out*).

Implémentation du module Stack

Le fichier stack_squel.py contient le squelette de l'implémentation d'une pile.

```
class StackEmptyError(Exception):
    """
    exception pour pile vide
    """
    def __init__(self, msg):
        self.message = msg

class Stack:
    """
    une classe pour manipuler les piles
    """

    def __init__(self):
        """
        constructeur de pile
        """
        pass

    def is_empty(self):
        """
        :return: (bool) True si la pile est vide, False sinon
        :CU: None
        :Exemples:

        >>> p = Stack()
        >>> p.is_empty()
        True
        >>> p.push(1)
        >>> p.is_empty()
        False
        """
        pass

    def push(self, el):
        """
        :param el: (any) un élément
        :return: None
        :Side-Effet: ajoute un élément au sommet de la pile
        :CU: None

        >>> p = Stack()
        >>> p.push(1)
```

```
        >>> p.pop() == 1
        True
        """
        pass

    def pop(self):
        """
        :return: (any) l'élément au sommet de la pile
        :CU: la pile ne doit pas être vide
        :raise: StackEmptyError
        :Side-Effect: la pile est modifiée
        :Exemples:

        >>> p = Stack()
        >>> p.push(1)
        >>> p.pop() == 1
        True
        >>> p.is_empty()
        True
        """
        pass

    def top(self):
        """
        :return: (any) l'élément au sommet de la pile
        :CU: la pile ne doit pas être vide
        :raise: StackEmptyError
        :Exemples:

        >>> p = Stack()
        >>> p.push(1)
        >>> p.top() == 1
        True
        >>> p.is_empty()
        False
        """
        pass

if __name__ == "__main__":
    import doctest
    doctest.testmode(verbose=True)
```

Renommez-le et implémentez votre propre structure de pile.

Enfin, implémenter l'algorithme de vérification dans un fichier `html_checker.py`.

Le squelette est fourni.

Amélioration du parser

Utiliser des expressions régulières

L'objectif est ici d'écrire une nouvelle version du parser permettant de prendre en compte l'ensemble des situations particulières listées plus haut.

Pour cela, nous pouvons utiliser une expression régulière. Une expression régulière est une chaîne de caractère spécifiant un format permettant de :

- dire si une chaîne correspond au format ;

- capturer certaines parties de ce format.

Un cours sur les expressions régulières est hors du programme du DIU, mais nous allons en fournir une pour récupérer les tags d'un document html.

Le code suivant contient une expression régulière présentée sur plusieurs lignes pour plus de lisibilité. Elle permet de récupérer les tags en tenant compte des guillemets.

```
import re

HTML_REGEX = re.compile(
    r"""<                          # commence par un <
(/?\w+)                            # capture le nom du tag (éventuellement avec un slash
(                                  # groupe des attributs
 (
  \s+                              # autant d'espace que l'on veut
  \w+                              # suivi d'un mot (nom de l'attribut)
  (
   \s*=\s*                         # un signe égal, entouré d'espaces
   (?:".*?"|'.*?'|[\^'">\s]+)      # valeur de l'attribut : capture de tout ce qui est entre guillemets
  )?                               # valeur optionnelle
 )+                                # autant de couple nom=valeur que l'on veut
 \s*                               # suivi d'éventuelles espaces
 |\s*                              # ou bien pas d'attributs : que des espaces
)                                  # fin des attributs
>                                  # finit par un >
""", re.VERBOSE)
html = """
<!DOCTYPE html>
<!-- un commentaire -->
<html lang="fr-FR">
  <body>
    <h1> Un titre </h1>
    <p class='code'>
         un paragraphe
         while i < len(l):
    </p>
    <br/>
  </body>
</html>
"""
for tag_element in HTML_REGEX.findall(html):
    print(tag_element[0])
```

```
html
body
h1
/h1
p
/p
/body
/html
```

En utilisant cette expression régulière, ou sa forme condensée (ci-dessous), écrire une deuxième version du parser : `html_parser2.py`.

Forme condensée de l'expression régulière :

```
HTML_REGEX = re.compile(
    r"""<(/?\w+)((\s+\w+(\s*=\s*(?:".*?"|'.*?'|[\^'">\s]+))?)+\s*|\s*)>""")
```

Pour aller plus loin : utiliser une classe existante

Utiliser une expression régulière a toujours ses limites pour parser un fichier html. Des bibliothèques existent pour itérer sur les tags d'un tel document.

Parmi elles, on peut utiliser la classe `HTMLParser`. Voici le code d'une classe héritant de `HTMLParser` et qui construit la liste des tags :

```
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    """
    a class that parse a document and allow tag access
    :exemples:

    >>> parser = MyHTMLParser("<!DOCTYPE hml><html lang="fr"></html>")
    >>> parser.has_tag()
    True
    >>> parser.next_tag()
    '<html>'
    >>> parser.next_tag()
    '</html>'
    >>> parser.has_tag()
    False
    """
    def __init__(self, data):
        """
        constructor for MyHTMLParse
        :param data: (str) html document
        :UC: None
        """
        super().__init__()
        self.__tags = []
        self.__tag_index = 0
        HTMLParser.feed(self, data)

    def handle_starttag(self, tag, attrs):
        """
        handle an opening tag
        :param tag: (str) the opening tag
        :param attrs: (list) attributes
        """
        self.__tags.append('<{:s}>'.format(tag))

    def handle_endtag(self, tag):
        """
        handle an ending tag
        :param tag: (str) the ending tag
        """
        self.__tags.append('</{:s}>'.format(tag))

    def has_tag(self):
        """
        :return: (bool) True if document contains another tag, False otherwise
        """
        return self.__tag_index < len(self.__tags)

    def next_tag(self):
```

```
        """
        :return: (str) the next tag in document
        """
        res = self.__tags[self.__tag_index]
        self.__tag_index += 1
        return res

if __name__ == '__main__':
    import doctest
    doctest.testmod(optionflags=doctest.NORMALIZE_WHITESPACE | doctest.ELLIPSIS, verbose=False)
```