TP sur les files et les piles

Thème 1

I Motivation

1 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 la page du W3C dédiée [1]. 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 [2] 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 [3] de fichier HTML, permettant de parcourir séquentiellement les balises.

II 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 `<!- -` et `- ->`
- 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.

1 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.

1. https://www.w3.org/community/webed/wiki/HTML/Training/Tag_syntax
2. **Prédicat** : dans ce contexte, il s'agit d'une fonction retournant `True` ou `False` en relation à une propriété du paramètre de cette dernière.
3. **parser** : Programme informatique qui met en évidence la structure d'un texte, d'un programme informatique.

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

Le fichier `myqueue.py` contient le squelette d'un module implémentant une structure de file (on ne peut utiliser le nom `queue.py` car Python dispose déjà d'un module homonyme).

A faire

1. Prendre connaissance du paragraphe 5.1.2 de la documentation Python [a] .
   Comme ce paragraphe l'indique : *toutefois, les listes ne sont pas très efficaces pour réaliser ce type de traitement. 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).*
   Aussi, il sera judicieux d'utiliser comme attribut de notre classe `Queue` une structure de la classe `collections.deque`.
2. Recopier le fichier `myqueue.py` dans votre espace de travail et compléter-le.

a. 5.1.2 Utilisation des listes comme des files : https://docs.python.org/fr/3/tutorial/datastructures.html#using-lists-as-queues

2 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_parser1.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` [4] des chaînes de caractères :

```
>>> help(str.index)
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` [5] peut également s'avérer utile.

A faire

Recopier le fichier `html_parser1.py` dans votre espace de travail et compléter-le en tenant des conseils ci-dessus.

III 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.

4. *Méthode index* : https://docs.python.org/fr/3/library/stdtypes.html?highlight=index#str.index
5. *Méthode split* : https://docs.python.org/3.6/library/stdtypes.html#str.split

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.

IV 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).

V Implémentation du module mystack

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

A faire

1. Copier dans votre espace de travail le fichier `mystack.py` et implémentez votre propre structure de pile.
2. Enfin, copier dans votre espace de travail le fichier `html_checker.py` et implémenter l'algorithme de vérification.

VI Amélioration du parser

1 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, mais nous allons en fournir une pour récupérer les tags d'un document html.

Le code du fichier `html_parser2.py` contient une expression régulière présentée sur plusieurs lignes pour plus de lisibilité. Elle permet de récupérer les tags en tenant compte des guillemets.

A faire

- Copier le fichier `html_parser2.py` dans votre espace de travail.
- Bien regarder l'exemple d'utilisation de l'expression régulière pour la comprendre (lignes 30 à 45).
  Ne pas hésiter à tester plusieurs exemples dans la console Python.
- En utilisant cette expression régulière, écrire une deuxième version du parser en complétant la fonction `parser`.

2 Pour aller plus loin

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`. Le fichier `html_parser3.py` contient le code d'une classe héritant de `HTMLParser` et qui construit la liste des tags.

A faire

1. Copier le fichier `html_parser3.py` dans votre espace de travail.
2. Compléter la fonction `parser` utilisant la classe décrite dans ce même fichier.

VII Source

Université de Lille (2020, 21 novembre). *Piles et files*. Enseignement DIU EIL.
https://gitlab-fil.univ-lille.fr/diu-eil-lil/portail/-/blob/master/bloc4-5/pile_file/pile.md

VIII Annexes

1 myqueue.py

```python
import collections

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
        """
        self.inside = #A compléter avec un objet vide de la classe "deque"

    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)
```


2 html_parser1.py

```
from myqueue import Queue

def parse(doc):
    """
    :param doc: (str) a document to be parsed
    :return: (Queue) a queue containing tag (without attributes)
    :CU: None
    :Example:

    >>> q = parse("<!DOCTYPE html><html><div class='titre'>du contenu ignoré</div></html>")
    >>> q.is_empty()
    False
    >>> q.dequeue() == "<html>"
    True
    >>> q.dequeue() == "<div>"
    True
    >>> q.dequeue() == "</div>"
    True
    >>> q.dequeue() == "</html>"
    True
    >>> q.is_empty()
    True
    >>> # Other example
    >>> q = parse("<html><div><p <a")
    >>> q.dequeue() == "<html>"
    True
    >>> q.dequeue() == "<div>"
    True
    >>> q.is_empty()
    True
    """
    pass

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


3 mystack.py

```
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.testmod(verbose=True)
```


4 html_checker.py

```python
from mystack import Stack
from myqueue import Queue
from html_parser1 import parse

def html_checker(s):
    """
    :param s: (str) un document html
    :return: (bool) True si `s` est bien formé, False sinon
    :CU: Aucune
    :Exemple:

    >>> html_checker("<!DOCTYPE html><html lang='fr'><div><p></p></div></html>")
    True
    >>> html_checker("<!DOCTYPE html><html lang='fr'><div></p></div><p></html>")
    False
    >>> html_checker("<!DOCTYPE html><html lang='fr'><div><p></p></div>")
    False
    """
    pass

if __name__ == "__main__":
    import sys
    if len(sys.argv) <= 1:
        # pas d'arguments sur la ligne de commande
        # on exécute les tests.
        import doctest
        doctest.testmod(verbose = True)
    else:
        # on considère que l'argument est le nom d'un fichier
        fname = sys.argv[1]
        try:
            with open(fname, 'r') as f:
                if html_checker(f.read()):
                    print("le fichier {} est un fichier html correct.".format(fname))
                else:
                    print("le fichier {} n'est pas un fichier html correct.".format(fname))
        except:
            print("Impossible de parser le fichier {}".format(fname))
```


5 html_parser2.py

```
from myqueue import Queue

import re

#Expression régulière
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)

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

#Exemple d'utilisation :
"""
>>> 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
"""
```

```python
def parse(doc):
    """
    :param doc: (str) a document to be parsed
    :return: (Queue) a queue containing tag (without attributes)
    :CU: None
    :Example:

    >>> q = parse("<!DOCTYPE html><html><div class='titre'>du contenu ignoré</div></html>")
    >>> q.is_empty()
    False
    >>> q.dequeue() == "html"
    True
    >>> q.dequeue() == "div"
    True
    >>> q.dequeue() == "/div"
    True
    >>> q.dequeue() == "/html"
    True
    >>> q.is_empty()
    True
    >>> # Other example
    >>> q = parse("<html><div><p <a")
    >>> q.dequeue() == "html"
    True
    >>> q.dequeue() == "div"
    True
    >>> q.is_empty()
    True
    """
    pass

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

6 html_parser3.py

```python
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

def parser(doc):
    pass

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