TD 4

1. Quelques rappels de syntaxe

- Fonctions utiles en mode interactif
  - `type(truc)`
  - `dir(truc)`

- Chaîne <=> Liste
  - `"mot1_mot2_mot3".split("_")` → `["mot1", "mot2", "mot3"]`
  - `"_".join(["mot1", "mot2", "mot3"])` → `"mot1_mot2_mot3"`
  - `regex.split(r"_", "mot1_mot2_mot3")` → `["mot1", "mot2", "mot3"]`

- Range ≠ Liste
  - `if 1 in ragne(0, 2): print("ok")`
  - `if 2 in range(0, 2): print("ok")`
  - `if 1 in [0, 1]: print("ok")`
  - `if 2 in [0, 1]: print("ok")`

2. Regex

2.1. Les regex en python

- `import regex` (PCRE mieux que `import re` qui n'est que POSIX)
- `r"\n"` ≠ `"\n"` <–mais–> `r"\n"` = `"\\n"`

2.1.1. Méthode *split*

`regex.split(r" ", string, maxsplit=0, flags=0)` → `list`

Tokenisez le fichier *volume07.txt* ligne par ligne, en convertissant chaque ligne en liste, de façon de + en + complète (commencez par le *1*, puis ajoutez les contraintes du *2*, etc.) :

1. uniquement sur les espaces
2. en tenant compte des signes de ponctuation mono-caractères (.,;!:?=)
3. les signes de ponctuation doivent être conservés comme tokens
   - extrait de la doc de la méthode *split* :

     > Split *string* by the occurrences of *pattern*. **If capturing parentheses are used in *pattern*, then the text of all groups in the pattern are also returned as part of the resulting list.**
4. en tenant compte des signes de ponctuation multi-caractères (…)
5. les séquences de 2 tirets ou + (-------) doivent être tokenisés ensemble

Métacaractères	
^	Début de chaîne ou début de ligne
$	Fin de chaîne ou fin de ligne
.	N'importe quel caractère sauf retour à la ligne
|	Disjonction (a|b pour a ou b)
[ ]	Marques de début et fin de classe
( )	Marques de début et fin de groupe

2

3 Ces caractères peuvent être déspécialisés avec le car. \

Opérateurs de répétition	
*	0 à n fois
+	1 à n fois
?	0 ou 1 fois
{3}	exactement 3 fois
{3,}	3 fois ou plus
{3,5}	3 à 5 fois

4

Les opérateurs portent sur le car. ou le groupe qui précède

Les opérateurs ont par défaut un comportement gourmand ("greedy"), ajoutez un ? pour les rendre "ungreedy"

2.1.2.Listes, *sets* et dictionnaires

Regroupez tous les tokens :

- dans une liste (la liste des tokens du texte)

Using the `append()` method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

Add the elements of `tropical` to `thislist`:

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)
```

- dans un *set* (le lexique du texte)

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Add elements from `tropical` into `thisset`:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

- dans un dictionnaire (token => fréquence)

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Add a color item to the dictionary by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

2.1.3.Méthode *match*

`regex.match(r" ", string)` → `Match` ou `None` si non trouvé

`re.`**`match`**`(`*`pattern, string, flags=0`*`)`

If zero or more characters at the beginning of *string* match the regular expression *pattern*, return a corresponding `Match`. Return `None` if the string does not match the pattern; note that this is different from a zero-length match.

2.1.4.Méthode search

`regex.search(r" ", string)` → `Match` ou `None` si non trouvé

`re.`**`search`**`(`*`pattern, string, flags=0`*`)`

Scan through *string* looking for the first location where the regular expression *pattern* produces a match, and return a corresponding `Match`. Return `None` if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.

Utilisation dans un *if* :

- ```
  if regex.search(r"abc", string):
      print("trouvé !")
  ```
- ```
  m = regex.search(r"abc", string)
  if m:
      print(m.group(0))
  ```

Écrire une fonction `is_ponct(token)` qui retourne True si le token est un signe de ponctuation.

Écrire une fonction `is_entity(token, nextToken)` qui retourne True si le token est une entité XML (p. ex. * *, *a*, *a*), sachant que du fait de la tokenisation, dans ce cas, le caractère ; sera dans le token suivant.

Écrire une fonction `is_num(token, nextToken)` qui retourne True si le token est un nombre arabe ou romain (caractères dans ivxlcdmj, où j ne peut apparaître qu'en dernière position, et où nextToken est un point).

Écrire une fonction `is_greek(token)` qui retourne True si le token est un mot grec.

Écrire une fonction is_lemma(token1, token2, token3) qui retourne le *Lemma* si token3 en est un, ou chaîne vide si ce n'en est pas un. Faire ensuite de même avec *Author* et *Normalized Classification*.

Pour aller plus loin…

Utilisez les lexiques fournis ici : https://gitlab.com/ANR-DFG-presto/prestoProfile/-/tree/master/lexicon (bouton "Télécharger le code source", puis dans l'archive obtenue aller dans le dossier *lexicon*). Fichiers lexPresto_20151027.csv (lexique du français classique) et latin-lexicon.csv (lexique du latin).

Écrire une fonction `read_lexicon(filename)` qui retourne un *set* du lexique. Attention, ces fichiers CSV comportent plusieurs colonnes délimitées par des tabulations, et la première ligne est un en-tête.

Utilisez le *set* ainsi obtenu pour déterminer, pour chaque token, si il est français ou latin (il peut être les deux).

3. Counter

A `Counter` is a `dict` subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The `Counter` class is similar to bags or multisets in other languages.

```
from collections import Counter
```

3.1. Création/ajout

```
c = Counter({'red': 4, 'blue': 2})      # a new counter from a mapping
```

update([*iterable-or-mapping*])

Elements are counted from an *iterable* or added-in from another *mapping* (or counter). Like `dict.update()` but adds counts instead of replacing them. Also, the *iterable* is expected to be a sequence of elements, not a sequence of `(key, value)` pairs.

3.2. Fonctions utiles

most_common([*n*])

Return a list of the *n* most common elements and their counts from the most common to the least. If *n* is omitted or `None`, `most_common()` returns *all* elements in the counter. Elements with equal counts are ordered in the order first encountered:

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

3.3. Exercices

Utilisez un *Count* pour afficher :

1. Les tokens les plus fréquents du texte.
2. Les mots français les plus fréquents du texte.
3. Les mots les plus fréquents de chaque *Normalized Classification.*