

Introduction aux expressions régulières

00:00 - 00:06

Bienvenue dans le cours ! Dans cette vidéo, vous allez apprendre ce que sont les expressions régulières.

2. Qu'est-ce que le traitement automatique du langage naturel (TALN) ?

00:06 - 00:47

Le traitement automatique du langage naturel (TALN) est un vaste domaine d'étude et une pratique largement utilisée, qui vise à donner un sens au langage en utilisant des statistiques et des ordinateurs.

Dans ce cours, vous apprendrez certaines bases du TALN qui vous aideront à progresser des sujets simples vers des sujets plus complexes et avancés.

Même si c'est le premier cours, vous aurez tout de même un aperçu de certains défis du domaine, tels que **l'identification de sujets et la classification de textes**.

Voici quelques domaines intéressants du TALN que vous avez peut-être déjà entendus :

- identification de sujets
- chatbots
- classification de textes
- traduction
- analyse de sentiments

Et il y en a encore bien d'autres !

Vous apprendrez les bases de certains de ces sujets au fil du cours.

3. Que sont exactement les expressions régulières ?

00:47 - 01:54

Les expressions régulières sont des chaînes de caractères avec une **syntaxe spéciale**, qui vous permettent de **rechercher des motifs (patterns)** et de retrouver d'autres chaînes de caractères. Un motif est une série de lettres ou de symboles qui peut correspondre à un texte réel, à des mots ou à de la ponctuation.

Vous pouvez utiliser les expressions régulières pour :

- trouver des liens dans une page web
- extraire des adresses e-mail
- supprimer des caractères ou chaînes non désirées

Les expressions régulières sont souvent appelées **regex** et sont faciles à utiliser en Python grâce à la bibliothèque `re`.

Ici, nous avons un simple import de la bibliothèque.

On peut comparer un sous-texte en utilisant la méthode `re.match`, qui permet de vérifier si un motif correspond à une **chaîne de caractères**.

Elle prend le motif comme premier argument, la chaîne comme deuxième argument, et retourne un **objet de correspondance** (*match object*).

Par exemple, on voit ici qu'elle a parfaitement trouvé ce qu'on attendait : `abc`.

On peut aussi utiliser des motifs spéciaux compris par regex, comme `\w+` qui correspond à un **mot entier**.

On voit ici, via l'objet match, qu'il a trouvé le **premier mot** qu'il a rencontré : `hi`.

4. Motifs regex courants

01:54 - 02:10

Il existe des **centaines de caractères et de motifs** que vous pouvez apprendre et mémoriser avec les expressions régulières.

Mais pour commencer, voici quelques motifs courants :

- `\w` : correspond à un **mot** (lettres ou chiffres)
- `\d` : correspond à un **chiffre**
- `\s` : correspond à un **espace**
- `.` (point) : correspond à **n'importe quel caractère**

Les symboles `+` et `*` rendent les motifs "gourmands" : ils prennent les répétitions d'un même caractère ou d'un motif entier.

Par exemple, pour trouver un mot entier au lieu d'un seul caractère, on ajoute le `+` après `\w`.

Utiliser ces classes de caractères en **majuscules** permet de **nier** la classe :

- `\S` correspond à **tout ce qui n'est pas un espace**

Vous pouvez aussi créer un **groupe de caractères personnalisé** en les plaçant entre crochets, comme pour un groupe de lettres minuscules.

5 à 10. Autres motifs regex courants (2 à 7)

02:10 - 03:02

(Le texte ne fournit pas le détail ici, ce sont probablement des exemples supplémentaires que tu verras dans la vidéo.)

11. Le module `re` de Python

03:02 - 03:55

Dans les exercices suivants, vous allez utiliser le module `re` pour effectuer des actions simples comme **séparer une chaîne selon un motif** ou **trouver tous les motifs** dans une chaîne.

En plus de `split` et `findall`, les méthodes `search` et `match` sont aussi très utilisées.

- `match` vérifie si le **début de la chaîne** correspond au motif
- `search` peut trouver le motif **n'importe où dans la chaîne**

La syntaxe de la bibliothèque regex est toujours :

👉 le motif en premier, puis la chaîne en second.

Selon la méthode utilisée, vous pouvez obtenir un :

- itérateur
- nouvelle chaîne
- objet de correspondance

Exemple : la méthode `re.split` peut prendre un motif d'espace et une chaîne avec des espaces, et retourner une liste contenant les mots séparés.

Cela peut être utilisé pour la **tokenisation** (couper le texte en mots), donc très utile pour prétraiter le texte dans un projet de traitement du langage naturel.

12. À vous de jouer !

03:55 - 04:02

Maintenant, c'est à vous !

Commencez à écrire votre première expression régulière et on se retrouve très vite !

Pas de souci ! Tu n'es pas seul à trouver ça difficile au début 😊

Les **expressions régulières** (ou **regex**) sont un peu comme une **langue spéciale pour chercher dans du texte**. On va prendre notre temps pour tout comprendre, avec des exemples simples et visuels.

🔍 1. Qu'est-ce qu'une expression régulière (regex) ?

C'est une façon d'écrire un "modèle" (pattern) pour dire à l'ordinateur :

👉 "Je veux trouver un texte qui ressemble à ceci."

📦 2. Exemple concret :

Imaginons qu'on a ce texte :

Bonjour ! Mon email est : julie.dupont123@gmail.com. Tu peux m'écrire. ⚡ Copy

🎯 Objectif : Extraire l'adresse email

Normalement, c'est difficile pour un ordinateur de deviner ce qu'est un email.

Mais avec une **expression régulière**, on peut lui dire :

\w+[\w\.]*@\w+\.\w+

⚡ Copy

🧠 Que signifie ce pattern ?

Symbol	Signification
\w	Une lettre ou un chiffre (ex: a , b , 1)
\w+	Un mot (au moins une lettre/chiffre)
[\w\.]*	Zéro ou plusieurs lettres, chiffres ou points
@	Le caractère arobase (obligatoire dans un email)
\w+	Le nom de domaine (gmail, yahoo, etc.)
\.	Un point
\w+	Le type de domaine (com, fr, org...)

👉 Cela permet de reconnaître des emails comme :

julie.dupont123@gmail.com
nabil_k@gmail.fr

⚡ Copy

🐍 3. Utilisation en Python :

Voici comment on peut faire ça avec Python et le module `re` :

```
import re

texte = "Bonjour ! Mon email est : julie.dupont123@gmail.com. Tu peux m'écrire."

# Pattern pour détecter une adresse email
pattern = r"\w+[\w\.] *@\w+\.\w+"

# Cherche tous les emails
emails = re.findall(pattern, texte)

print(emails)
```

 Copy

📋 Résultat :

```
['julie.dupont123@gmail.com']
```

 Copy

▀ 4. Autres cas d'usage fréquents :

Exemple de texte	Objectif	Regex (pattern)	Résultat
"Il a 23 ans."	Trouver les nombres	\d+	23
"Code postal : 75001"	Extraire le code postal	\d{5}	75001
"Tel: 06-12-34-56-78"	Extraire numéro de tel	\d{2}-\d{2}-\d{2}-\d{2}-\d{2}	06-12-34-56-78

⚠ 5. Attention :

Les regex ne lisent pas comme des phrases, elles sont logiques et symboliques. C'est comme une formule mathématique pour du texte.

 **En résumé simple :**

But	Regex
Trouver des chiffres	\d+
Trouver des mots	\w+
Trouver des espaces	\s+
Trouver un email	\w+[\w\.\.]*@\w+\.\w+

Cet exercice te permet de pratiquer les expressions régulières (`regex`) en utilisant deux fonctions importantes du module `re` de Python :

- `re.split()` → pour découper une chaîne de caractères selon un motif.
- `re.findall()` → pour trouver tous les éléments qui correspondent à un motif dans une chaîne.

🔍 Contexte

Avant de plonger dans l'explication ligne par ligne, voici ce qu'on essaie de faire :

On part d'une chaîne de texte nommée `my_string` (que tu peux afficher dans l'IPython Shell avec `print(my_string)`).

Le but est d'y chercher :

- des fins de phrase
- des mots commençant par une majuscule
- des espaces
- des chiffres

📘 Explication ligne par ligne

```
# Write a pattern to match sentence endings: sentence_endings
sentence_endings = r"[.?!]"
```

 Copy

👉 Que fait cette ligne ?

- Le motif `[.?!]` dit : "trouve tous les caractères qui sont un point `.`, un point d'exclamation `!` ou un point d'interrogation `?`."
- Ces caractères sont utilisés pour marquer la fin d'une phrase.

```
print(re.split(sentence_endings, my_string))
```

 Copy

👉 Que fait cette ligne ?

- Elle utilise `re.split()` pour découper la chaîne `my_string` à chaque fin de phrase (repérée par le motif `[.?!]`).
- Tu obtiens une liste de phrases.

```
capitalized_words = r"[A-Z]\w+"
print(re.findall(capitalized_words, my_string))
```

 Copy

👉 Que fait cette ligne ?

- Le motif `[A-Z]\w+` dit :

"Trouve un mot qui commence par une majuscule suivie d'une ou plusieurs lettres/chiffres."

- `re.findall()` renvoie tous les mots commençant par une majuscule (souvent des noms propres ou débuts de phrases).

```
spaces = r"\s+"
print(re.split(spaces, my_string))
```

 Copy

👉 Que fait cette ligne ?

- Le motif `\s+` correspond à un ou plusieurs espaces blancs (`\s` = espace, tabulation, retour à la ligne, etc.).
- Cela permet de découper `my_string` en mots individuels.

```
digits = r"\d+"
print(re.findall(digits, my_string))
```

 Copy

👉 Que fait cette ligne ?

- Le motif `\d+` signifie "trouve un ou plusieurs chiffres consécutifs".
- Cela permet de repérer tous les nombres dans `my_string`.

⚠️ À retenir sur les chaînes "raw" (`r""`)

Quand tu écris une regex, il faut souvent la précéder de `r` (pour "raw string"). Sinon, Python pourrait mal interpréter certains caractères spéciaux.

Par exemple :

```
"\n" → saut de ligne
r"\n" → deux caractères : "\n" et "n" (ce qu'on veut en regex)
```

 Copy

Résumé visuel :

Objectif	Regex	Fonction utilisée
Couper à la fin des phrases	[.?!]	re.split()
Trouver les mots majuscules	[A-Z]\w+	re.findall()
Couper par mot (espaces)	\s+	re.split()
Trouver des chiffres	\d+	re.findall()

Si tu veux, je peux t'écrire un exemple de `my_string` pour tester ces regex et mieux voir les résultats 

Tu veux ? 

Comprendre la Tokenisation avec NLTK & Regex

Voici la traduction en français de la vidéo, avec des explications supplémentaires et des exemples concrets pour bien comprendre la tokenisation en Python, notamment avec `nltk` et les expressions régulières.

1. Introduction à la tokenisation

⌚ 00:00 - 00:04

Dans cette vidéo, nous allons découvrir ce qu'est la **tokenisation de chaînes de caractères** !

2. Qu'est-ce que la tokenisation ?

⌚ 00:04 - 00:39

La **tokenisation** est le processus qui consiste à transformer une chaîne de caractères (ou un document entier) en éléments plus petits, appelés **tokens**.

Ce processus est une étape essentielle dans le **prétraitement des textes** pour le traitement du langage naturel (NLP).

→ Un token peut être :

- un mot
- une phrase
- une ponctuation
- un hashtag
- une mention (@)

Tout dépend de la règle de tokenisation utilisée.

🔧 Exemple 1 – Tokenisation manuelle (basique) :

```
text = "Je suis étudiant en IA. J'adore Python !"  
tokens = text.split() # Découpe par espaces  
print(tokens)
```

 Copy

🟢 Résultat :

```
['Je', 'suis', 'étudiant', 'en', 'IA.', "J'adore", 'Python', '!']
```

 Copy

⚠️ Ce n'est pas très précis : la ponctuation reste attachée à certains mots.

3. 📚 La bibliothèque `nltk`

⌚ 00:39 - 00:58

Une des bibliothèques les plus utilisées pour tokeniser des textes est **NLTK** (*Natural Language Toolkit*).

🔧 **Exemple 2 – Tokenisation avec `nltk` :**

```
from nltk.tokenize import word_tokenize

text = "Je suis étudiant en IA. J'adore Python !"
tokens = word_tokenize(text)
print(tokens)
```

Copy

🟢 Résultat :

```
['Je', 'suis', 'étudiant', 'en', 'IA', '.', 'J\'', 'adore', 'Python', '!']
```

Copy

✓ Les mots et la ponctuation sont séparés proprement.

4. 😐 Pourquoi tokeniser ?

⌚ 00:58 - 01:31

La tokenisation est **utile** pour de nombreuses tâches en NLP :

- Analyser la structure grammaticale (par exemple, identifier les verbes, noms, etc.)
- Filtrer les mots courants (comme "le", "de", "à" en français)
- Déetecter des éléments de sens, comme la négation ou la possession.

🔧 **Exemple 3 – Comprendre la structure :**

Phrase : "Je n'aime pas les chaussures de Sam."

Tokens :

```
['Je', 'n', "'", 'aime', 'pas', 'les', 'chaussures', 'de', 'Sam', '.']
```

Copy

- 'n' + 'pas' = **négation**
- 'de Sam' = **possession**

Ces informations peuvent être utiles pour interpréter le sens.

5. Autres tokenizers de nltk

⌚ 01:31 - 02:03

nltk propose plusieurs outils utiles :

Fonction	Description
sent_tokenize()	Découpe un texte en phrases
regexp_tokenize()	Utilise des expressions régulières pour tokeniser (très flexible)
TweetTokenizer()	Idéal pour les tweets : reconnaît hashtags, mentions (@), émoticônes, etc.

🔧 Exemple 4 – sent_tokenize() :

```
from nltk.tokenize import sent_tokenize  
  
text = "Bonjour. Comment vas-tu ? Voici un test."  
sentences = sent_tokenize(text)  
print(sentences)
```

Copy

🟢 Résultat :

```
['Bonjour.', 'Comment vas-tu ?', 'Voici un test.']}
```

Copy

6. 🔎 Pratique des regex avec re.match() et re.search()

⌚ 02:03 - 03:07

Les **expressions régulières (regex)** sont **essentielles** pour créer tes propres règles de tokenisation ou analyser du texte.

Fonction Python	Ce qu'elle fait
re.match()	Cherche un motif au début de la chaîne
re.search()	Cherche un motif n'importe où dans la chaîne

🔧 Exemple 5 – Différence entre match et search :

```
import re  
  
text = "abcde"  
print(re.match("abc", text))      # Match trouvé car c'est au début  
print(re.search("cde", text))     # Match trouvé même si ce n'est pas au début
```

Copy

```
print(re.match("cde", text))      # Aucun match
```

● Résultat :

```
<_sre.SRE_Match object; span=(0, 3), match='abc'>
<_sre.SRE_Match object; span=(2, 5), match='cde'>
None
```

 Copy

- Utilise `search()` si le motif peut être n'importe où dans la chaîne.
- Utilise `match()` si le motif doit être au tout début.

7. À toi de jouer !

⌚ 03:07 - 03:11

C'est le moment de pratiquer toi-même la tokenisation avec `nltk` et les expressions régulières !

FICHE MÉMO : `import` VS `from ... import ...` en Python



`import module`

Importe tout le module

- ◆ Tu dois utiliser le nom du module à chaque appel.

```
import nltk
tokens = nltk.tokenize.word_tokenize("Bonjour !")
```

Copy

Avantages :

- Plus clair quand tu lis le code.
- Moins de collisions de noms.



`from module import fonction`

Importe directement une ou plusieurs fonctions/classes

- ◆ Tu peux les utiliser directement sans préfixe.

```
from nltk.tokenize import word_tokenize
tokens = word_tokenize("Bonjour !")
```

Copy

Avantages :

- Code plus court
- Pratique pour une ou deux fonctions spécifiques



`as` pour créer un alias

Permet de raccourcir ou d'éviter des conflits de noms :

```
import numpy as np
import pandas as pd

print(np.array([1, 2, 3]))
```

Copy

Attention aux erreurs !

Mauvais code	Pourquoi c'est faux ?
<code>from nltk import word_tokenize</code>	<code>word_tokenize</code> n'est pas dans <code>nltk</code> , mais dans <code>nltk.tokenize</code>
<code>word_tokenize("texte")</code> (sans import)	La fonction n'est pas définie

Résumé Express

Situation	Syntaxe
Tout importer	<code>import module</code>
Importer une fonction spécifique	<code>from module.submodule import fonction</code>
Importer avec alias	<code>import module as alias</code>

Exemples utiles

```
# 1. Importer tout le module
import math
print(math.sqrt(16))

# 2. Importer juste une fonction
from math import sqrt
print(sqrt(16))

# 3. Avec alias
import matplotlib.pyplot as plt
```

 Copy

Astuce

 Quand tu débutes ou travailles sur de petits scripts :
`from ... import ...` est plus rapide.

 Pour les gros projets :
`import module` est plus sûr et lisible.

Parfait, on va analyser **ligne par ligne** ce petit script Python avec `nltk` (Natural Language Toolkit) et expliquer ce qu'on cherche à faire exactement à chaque étape 

Code à expliquer :

```
# Import necessary modules
from nltk.tokenize import sent_tokenize, word_tokenize
```

 Copy

Ce qu'on fait ici :

On importe deux fonctions du module `nltk.tokenize` :

- `sent_tokenize()` → pour **diviser un texte en phrases**
- `word_tokenize()` → pour **diviser une phrase ou un texte en mots/tokens**

```
sentences = sent_tokenize(scene_one)
```

 Copy

Ce qu'on fait ici :

On prend un texte appelé `scene_one` (il doit contenir un dialogue ou un texte complet) et on le **découpe en phrases**.

Chaque élément de la liste `sentences` sera une phrase du texte.

Par exemple :

`scene_one` → "Hello there! Who are you? I'm Arthur."

 `sentences` → ["Hello there!", "Who are you?", "I'm Arthur."]

```
print("Sentences:", sentences)
```

 Copy

Ce qu'on fait ici :

On affiche la **liste des phrases** qu'on vient d'extraire depuis `scene_one`.

```
tokenized_sent = word_tokenize(sentences[3])
```

 Copy

Ce qu'on fait ici :

On prend la **4e phrase** de la liste `sentences` (`sentences[3]`) et on la **découpe en mots (tokens)**.

Le résultat est une liste de mots, ponctuations incluses.

💡 Exemple :

Si `sentences[3]` = "ARTHUR: It is I, Arthur, son of Uther Pendragon..."

alors `tokenized_sent` sera →

`['ARTHUR', ':', 'It', 'is', 'I', ',', 'Arthur', ',', 'son', 'of', 'Uther', 'Pendr`

```
unique_tokens = set(word_tokenize(scene_one))
```

 Copy

🧠 Ce qu'on fait ici :

- On découpe **tout le texte** `scene_one` en mots avec `word_tokenize()`
- Puis on le convertit en `set()` pour ne garder que les **mots uniques**, c'est-à-dire éliminer **les doublons**.

💡 Exemple :

Si `word_tokenize(scene_one)` donne

`['Hello', 'Hello', 'Arthur', 'Arthur', '!']`

alors `set(...)` donne → `{'Hello', 'Arthur', '!'}`

```
print(unique_tokens)
```

 Copy

🧠 Ce qu'on fait ici :

On affiche tous les **mots uniques** présents dans le texte `scene_one`.

✍ Résumé global

Étape	Ce qu'on fait
1	On importe deux fonctions de tokenisation depuis NLTK.
2	On divise le texte en phrases .
3	On affiche les phrases pour les lire.
4	On prend une phrase spécifique (la 4e) et on la divise en mots/tokens .
5	On extrait tous les mots uniques du texte entier.
6	On affiche ces mots uniques.

Explication du Code : Recherche du mot "coconuts"

```
# Search for the first occurrence of "coconuts" in scene_one: match  
match = re.search("coconuts", scene_one)
```

 Copy

Ce qu'on fait ici :

On utilise la fonction `re.search()` du module `re` (pour les expressions régulières) afin de rechercher la première occurrence du mot "coconuts" dans le texte `scene_one`.

La fonction retourne un objet `match` contenant des informations sur l'endroit où le mot est trouvé dans le texte, comme son indice de début et de fin.

- Si le mot est trouvé, un objet `match` est retourné.
- Si le mot n'est pas trouvé, `match` sera égal à `None`.

```
# Print the start and end indexes of match  
print(match.start(), match.end())
```

 Copy

Ce qu'on fait ici :

On affiche les indices de début et de fin de la première occurrence du mot "coconuts" dans le texte.

- `match.start()` renvoie l'indice du début du mot "coconuts".
- `match.end()` renvoie l'indice juste après la fin du mot "coconuts" (c'est-à-dire l'indice du premier caractère après le mot).

Exemple

Si `scene_one` est "Arthur: You're using coconuts!" et que le mot "coconuts" commence à l'indice 22 et se termine à l'indice 31, alors :

- `match.start()` retournera 22
- `match.end()` retournera 31

Attention

Si le mot n'est pas trouvé, cela pourrait entraîner une erreur (car on tente d'accéder aux méthodes `start()` et `end()` sur un objet `None`). Il est donc recommandé de vérifier si `match` est `None` avant d'appeler ces méthodes.

Explication du Code : Recherche de Texte entre Crochets

```
# Write a regular expression to search for anything in square brackets: ⚡ Copy
pattern1
pattern1 = r"\[.*\]"
```

Ce qu'on fait ici :

On définit une **expression régulière** (regex) qui permet de rechercher **tout texte** contenu dans des **crochets carrés** (`[]`).

L'expression régulière `r"\[.*\]"` se décompose comme suit :

- `r""` : Le préfixe `r` indique qu'il s'agit d'une **chaîne brute** (raw string), ce qui permet de ne pas avoir à échapper certains caractères spéciaux comme `\`.
- `\[` : Le caractère `[` est un caractère spécial en regex, il doit donc être échappé avec un `\` pour le chercher littéralement.
- `.*` : Le point `.` signifie "n'importe quel caractère", et l'astérisque `*` signifie "zéro ou plusieurs occurrences" de ce caractère. Cela permet de capturer tout ce qui se trouve entre les crochets.
- `\]` : De la même manière que le `[` au début, le `]` doit également être échappé pour le rechercher littéralement.

Ainsi, cette regex recherche **n'importe quel texte entre crochets** (y compris des espaces et des caractères spéciaux).

```
# Use re.search to find the first text in square brackets
print(re.search(pattern1, scene_one)) ⚡ Copy
```

Ce qu'on fait ici :

On utilise la fonction `re.search()` pour **chercher la première occurrence** de texte qui correspond à la **regex** définie par `pattern1` dans la chaîne `scene_one`.

- Si un texte entre crochets est trouvé, `re.search()` retourne un objet **match** contenant des informations sur la correspondance.
- Si aucun texte entre crochets n'est trouvé, la fonction retourne `None`.

Le résultat est donc un **objet match** qui nous donne le premier texte trouvé entre crochets dans le texte `scene_one`.

Exemple

Si `scene_one` est `"King Arthur [wind] rode to Camelot [clop clop]"`, la regex cherchera le premier texte entre crochets, donc ici `"wind"`.

Le résultat serait quelque chose comme :

```
<re.Match object; span=(12, 18), match='[wind]'>
```

 Copy

Cela indique que le texte [wind] se trouve entre les indices 12 et 18 du texte scene_one .

Remarque

- `re.search()` ne retourne que la **première correspondance**. Si tu veux trouver toutes les occurrences, tu devras utiliser `re.findall()` à la place.

1. Comment retenir les patterns en regex, surtout avec `+` ou sans `+`
2. Et savoir ce que représente `re` exactement.

1. `re` est quoi exactement ?

`re` est un module standard de Python.

Il fait partie de la bibliothèque standard, donc pas besoin de l'installer : tu peux juste faire `import re`.

Le module `re` permet de travailler avec les expressions régulières (regex), pour rechercher, extraire, remplacer des motifs dans du texte.

2. Comment mémoriser les patterns ? (Surtout avec ou sans `+`)

Les caractères comme `+`, `*`, `?` sont appelés quantificateurs en regex.

Voici une petite fiche mentale très simple :

Symbol	Signification	Exemple	À retenir comme...
<code>*</code>	0 ou plusieurs fois	<code>a*</code> → "", "a", "aaaa"	aucun ou plein
<code>+</code>	1 ou plusieurs fois	<code>a+</code> → "a", "aaa"	au moins un
<code>?</code>	0 ou 1 fois	<code>a?</code> → "", "a"	optionnel

Exemple avec ton code :

```
pattern = r"\w+[\w\.]*@\w+\.\w+"
```

Copy

Décomposons ce pattern :

Partie	Explication
<code>\w+</code>	une ou plusieurs lettres/chiffres (le nom avant le <code>@</code>)
<code>[\w\.]*</code>	0 ou plusieurs lettres, chiffres ou points (ex: <code>.dupont123</code>)
<code>@</code>	le <code>@</code> littéral
<code>\w+</code>	une ou plusieurs lettres pour le nom de domaine (ex: <code>gmail</code>)
<code>\.</code>	un point littéral (doit être échappé)
<code>\w+</code>	une ou plusieurs lettres pour l'extension (ex: <code>com</code>)

Astuces pour mémoriser les patterns

1. Visualise le texte que tu veux capturer, ligne par ligne.
2. Crée ton regex comme si tu écrivais un “motif” de puzzle :
 - les lettres ? → `\w`
 - les chiffres ? → `\d`
 - au moins une fois ? → `+`
 - peut-être plusieurs fois ? → `*`
3. Utilise regex101.com (ou Pythex) pour tester et comprendre en live.
4. Pratique avec des cas du quotidien : emails, dates, numéros de téléphone, etc.

En résumé :

- `re` est un **module** (inclus dans Python).
- Le symbole `+` = 1 ou plusieurs fois, `*` = 0 ou plusieurs fois.
- Pour bien mémoriser : dissèque les **cas pratiques** comme celui de l'email que tu viens de faire.
- Tu peux faire une **fiche de patterns courants** (je peux t'aider à en faire une si tu veux !).

Tu veux que je t'aide à créer une fiche des patterns classiques (emails, dates, numéros, mots, etc.) ?

Parfait que tu poses la question 🎉 !

La **capturation de groupes** en regex, c'est un concept très utile... et aussi un peu flou au début 😊. Je vais t'expliquer simplement avec des images mentales et des exemples concrets.

🎬 Imagine que tu filmes une scène :

Tu veux garder **certaines parties** précises du film — des extraits, comme dans un montage vidéo.

En regex, les **parenthèses** () servent à "filmer" ces morceaux précis → ce sont les **groupes capturés**.

🔍 Exemple simple

```
import re
texte = "Mon numéro est : 06-12345678"
pattern = r"(\d{2})-(\d{8})"
match = re.search(pattern, texte)
```

 Copy

🧠 Décomposition du pattern :

- (\d{2}) → capture les **2 premiers chiffres** (le préfixe : "06")
- - → le tiret est juste **littéral**
- (\d{8}) → capture les **8 chiffres suivants**

✅ Résultat :

```
print(match.group(0)) # => '06-12345678' (tout le match)
print(match.group(1)) # => '06'           (1er groupe capturé)
print(match.group(2)) # => '12345678' (2e groupe capturé)
```

 Copy

💥 Maintenant sans parenthèses :

```
pattern = r"\d{2}-\d{8}"
match = re.search(pattern, texte)
print(match.group(0)) # => '06-12345678'
```

 Copy

Mais si tu fais :

```
print(match.group(1)) # ❌ ERREUR !
```

 Copy

Parce que tu n'as pas capturé de groupe ! 😬

📌 À quoi ça sert les groupes capturés ?

- Extraire une **partie spécifique** du texte (ex : le prénom, le domaine email, l'extension...)
- Remplacer **juste une partie** avec `re.sub()`
- Réutiliser ce qu'on a capturé dans une autre regex (avec `\1`, `\2`, etc.)

⟳ Petit exemple concret avec un e-mail

```
texte = "Contact : julie.dupont123@gmail.com"
pattern = r"([\w\.\.]+)@(\w+)\.( \w+)"
match = re.search(pattern, texte)

print("Nom :", match.group(1))      # julie.dupont123
print("Domaine :", match.group(2))   # gmail
print("Extension :", match.group(3)) # com
```

 Copy

✓ Grâce aux parenthèses, on **capture séparément** chaque partie de l'e-mail.

🧠 Résumé simple

Syntaxe	Ce que ça fait
<code>()</code>	Capture un groupe
<code>.group(0)</code>	Le texte entier qui correspond
<code>.group(1)</code>	Le 1er groupe capturé
<code>.group(2)</code>	Le 2e groupe , etc.

Tu veux que je t'entraîne avec des **petits défis de regex avec groupes capturés** ? Ou je te prépare une **fiche récap des cas d'usage** ?

Fiche Regex – Caractères Spéciaux et Comment les Chercher

◆ Quand tu veux chercher un caractère spécial en regex, il faut l'échapper avec un backslash `\`.

Liste des caractères spéciaux et comment les utiliser littéralement

Caractère	Signification en regex	Si tu veux le chercher au sens littéral, écris...
.	N'importe quel caractère sauf saut de ligne (<code>\n</code>)	<code>\.</code>
?	0 ou 1 fois	<code>\?</code>
+	1 ou plusieurs fois	<code>\+</code>
*	0 ou plusieurs fois	<code>*</code>
^	Début de ligne	<code>\^</code>
\$	Fin de ligne	<code>\\$</code>
[]	Classe de caractères	<code>\[\]</code>
()	Groupe capturant	<code>\()</code> et <code>\()</code>
{ } . . .	Répétitions (ex : <code>{2,4}</code>)	<code>\{ </code> et <code>\}</code> OU logique
\	Caractère d'échappement	<code>\\\</code>

Exemples concrets

1. Trouver un point dans un texte :

```
import re
texte = "Ceci est une phrase."
print(re.findall(r"\.", texte)) # Résultat : [']
```

 Copy

2. Déetecter un point d'interrogation :

```
texte = "Tu vas bien ?"  
print(re.findall(r"\?", texte)) # Résultat : ['?']
```

 Copy

3. Trouver un numéro de version, comme :

```
texte = "Dernière version : v2.5"  
print(re.findall(r"v\d\.\d", texte)) # Résultat : ['v2.5']
```

 Copy

Astuce : toujours utiliser les raw strings

- En Python, écris toujours ta regex comme : `r"\."` et non `"\."`
- Sinon, Python va interpréter `\.` comme un caractère d'échappement de chaîne avant même que la regex fonctionne.

En résumé

- Les caractères spéciaux en regex sont puissants... mais piégeux.
- Quand tu veux les utiliser **en tant que caractères normaux**, pense à mettre un `\` devant.
- Utilise toujours le format `r"..."` en Python.

Comprendre l'échappement des caractères spéciaux en Regex

Exactement, tu as bien compris ! 😊

Explication détaillée :

1. Les caractères spéciaux en regex doivent être échappés (avec un antislash `\`) pour être utilisés littéralement.

Par exemple :

- o Le point `.` en regex signifie "n'importe quel caractère" sauf un saut de ligne. Mais si tu veux rechercher un point littéral, comme un vrai `.` dans une adresse web ou un nom de fichier, tu dois l'échapper avec un antislash `\.`.
- o Le plus `+` signifie "une ou plusieurs répétitions". Si tu veux chercher le caractère plus lui-même dans une chaîne, tu dois l'échapper avec `\+.`.

Exemple pour un point :

```
import re
text = "Voici un texte avec un point."
result = re.findall(r"\.", text)
print(result) # ['.]
```

 Copy

2. Les caractères non spéciaux en regex (comme les lettres et chiffres) sont utilisés tels quels, sans besoin d'échappement.

Par exemple :

- o `a` dans une regex correspond simplement à la lettre "a". Tu n'as pas besoin d'échapper la lettre.
- o `0-9` ou `\d` correspond à un chiffre, et tu n'as pas besoin de l'échapper non plus.

Exemple pour rechercher un "a" :

```
import re
text = "Voici un texte avec un a."
result = re.findall(r"a", text)
print(result) # ['a', 'a']
```

 Copy

Récapitulatif :

- Caractères spéciaux en regex, comme `.` (point), `*` (étoile), `+` (plus), etc., ont des significations particulières dans le contexte des expressions régulières.
 - o Si tu veux rechercher ces caractères littéraux dans une chaîne, tu dois les échapper avec un antislash `\`.

Par exemple, pour chercher un **point littéral**, tu utiliseras `\.` au lieu de `.`.

- **Les caractères non spéciaux** (comme les lettres, chiffres, espaces, etc.) sont utilisés normalement dans les regex sans être échappés. Ils n'ont pas de signification particulière et sont donc pris tels quels.

Exemple combiné :

Si tu veux chercher un texte qui contient un numéro de téléphone avec des tirets (par exemple "06-12345678"), le modèle regex pourrait ressembler à ceci :

```
import re
text = "Mon numéro est 06-12345678"
pattern = r"\d{2}-\d{8}" # \d pour chiffres et le tiret '-' est littéral
result = re.findall(pattern, text)
print(result) # ['06-12345678']
```

Copy

Dans cet exemple :

- `\d{2}` : Recherchera **deux chiffres**.
- `-` : Recherchera un **tiret littéral**, donc il n'est pas échappé car il n'est pas spécial dans ce contexte.
- `\d{8}` : Recherchera **huit chiffres**.

Liste des caractères spéciaux en expressions régulières (regex)

Voici une liste des **caractères spéciaux** que l'on trouve en **expressions régulières (regex)**. Ces caractères ont des significations particulières et sont utilisés pour définir des motifs complexes.

💡 Liste des caractères spéciaux en regex :

1. Point (.)

- **Signification** : Correspond à **n'importe quel caractère**, sauf le saut de ligne (`\n`).
- Exemple : `a.b` correspond à "aab", "acb", mais **pas** "ab".

2. Antislash (\)

- **Signification** : Utilisé pour **échapper** un caractère spécial afin de le faire correspondre littéralement.
- Exemple : `\.` correspond au point littéral `.` et non à n'importe quel caractère.

3. Astre (*)

- **Signification** : Correspond à **zéro ou plusieurs occurrences** de l'expression précédente.
- Exemple : `a*` correspond à "", "a", "aa", "aaa", etc.

4. Plus (+)

- **Signification** : Correspond à **une ou plusieurs occurrences** de l'expression précédente.
- Exemple : `a+` correspond à "a", "aa", "aaa", mais **pas** "".

5. Interrogation (?)

- **Signification** : Correspond à **zéro ou une occurrence** de l'expression précédente (rendant l'élément **optionnel**).
- Exemple : `a?` correspond à "" ou "a".

6. Accolades ({})

- **Signification** : Spécifie un nombre **exact** d'occurrences de l'expression précédente.
- Exemple : `a{3}` correspond à "aaa" (exactement 3 "a").
- Exemple : `a{2,4}` correspond à "ad", "aad" ou "aaaa" (entre 2 et 4 occurrences).

7. Parenthèses (())

- **Signification** : Utilisé pour **capturer** un groupe d'expressions.
- Exemple : `(ab)+` correspond à "ab", "abab", "ababab", etc.
- Cela permet de **capturer** des groupes pour les réutiliser avec `group(1)`, `group(2)`, etc.

8. Crochets ([])

- **Signification** : Correspond à **un seul caractère** parmi une liste de caractères.
- Exemple : `[aeiou]` correspond à n'importe quelle voyelle.
- Exemple : `[0-9]` correspond à un chiffre entre 0 et 9.

9. Tiret dans les crochets (-)

- **Signification** : Définit une **plage de caractères**.
- Exemple : `[a-z]` correspond à une lettre minuscule, et `[0-9]` correspond à un chiffre.

10. Chapeau (^)

- **Signification** : Dans un **groupe de caractères**, il signifie **négation** (exclure un caractère spécifique).
- Exemple : `[^a-z]` correspond à **tout caractère** qui n'est pas une lettre minuscule.
- **Au début d'une regex**, il signifie **début de la chaîne**.
- Exemple : `^a` correspond à "a" au début de la chaîne.

11. Dollar (\$)

- **Signification** : Correspond à **la fin de la chaîne**.
- Exemple : `a$` correspond à "a" à la fin de la chaîne.

12. Barre verticale (|)

- **Signification** : OU logique. Elle permet de séparer plusieurs alternatives dans une regex.
- Exemple : `a|b` correspond à "a" ou "b".

13. Saut de ligne (\n)

- **Signification** : Correspond à un **saut de ligne**.
- Exemple : `a\nb` correspond à "a", suivi d'un saut de ligne, puis "b".

14. Tabulation (\t)

- **Signification** : Correspond à une **tabulation**.
- Exemple : `\ta` correspond à une tabulation suivie de "a".

15. Caractère d'échappement (\)

- **Signification** : Utilisé pour échapper des caractères spéciaux ou pour indiquer des caractères spéciaux (comme `\d`, `\w`, etc.).

✳ Quelques autres symboles importants en regex :

1. `\d` : Correspond à **un chiffre** (équivalent de `[0-9]`).
2. `\w` : Correspond à **un caractère alphanumérique** (lettres, chiffres, ou underscore).
Equivalent de `[a-zA-Z0-9_]` .
3. `\s` : Correspond à **un caractère d'espace blanc** (espace, tabulation, nouvelle ligne, etc.).
4. `\b` : Correspond à **une limite de mot** (avant ou après une séquence de caractères alphanumériques).
5. `\B` : Correspond à **une position qui n'est pas une limite de mot**.
6. `\W` : Correspond à un caractère **non alphanumérique** (tout ce qui n'est pas `[a-zA-Z0-9_]`).
7. `\S` : Correspond à **tout caractère qui n'est pas un espace blanc**.

📌 Résumé des caractères spéciaux les plus utilisés :

Caractère	Signification
.	N'importe quel caractère (sauf saut de ligne)
*	0 ou plusieurs occurrences
+	1 ou plusieurs occurrences
?	0 ou 1 occurrence (optionnel)
[]	Liste de caractères
,	,
()	Capture un groupe
^	Début de chaîne (ou négation dans [])
\$	Fin de chaîne
\	Caractère d'échappement

Les **parenthèses** et les **crochets** sont souvent les plus utilisés dans les expressions régulières pour capturer des groupes et définir des ensembles. Et comme tu l'as vu, certains caractères doivent être échappés (comme le point . ou l'astérisque *), car ils sont **spéciaux**.

D'accord, reprenons les choses plus lentement et avec des exemples plus simples pour comprendre les **caractères spéciaux en regex**.

1. Pourquoi les caractères spéciaux sont-ils utilisés ?

Les **expressions régulières (regex)** sont des modèles (ou motifs) utilisés pour rechercher et manipuler des chaînes de caractères (texte). Les **caractères spéciaux** en regex ont des significations particulières qui permettent de créer des recherches avancées, de filtrer et de modifier des parties spécifiques d'une chaîne.

2. Exemple simple d'expressions régulières :

Imaginons que tu veuilles rechercher des chiffres dans une chaîne de texte. Tu pourrais utiliser `\d` qui signifie **tout chiffre** (de 0 à 9).

Exemple :

```
import re
text = "Voici un numéro: 12345"
result = re.findall(r"\d", text)
print(result)
```

 Copy

Résultat : `['1', '2', '3', '4', '5']`

- Le modèle `\d` permet de capturer **chaque chiffre** dans la chaîne.

3. Les caractères spéciaux les plus courants en regex :

1. Point (.)

- Signification : Le **point** représente **n'importe quel caractère** sauf un saut de ligne.
- Exemple :

```
text = "hello world"
result = re.findall(r"h.ll.", text)
print(result) # ["hello"]
```

 Copy

Ici, le point `.` correspond à n'importe quel caractère entre `h` et `llo`, donc "hello" est trouvé.

2. Astre (*)

- Signification : L'**étoile** `*` signifie **zéro ou plusieurs répétitions** de ce qui précède.
- Exemple :

```
text = "aaaa bbb ab"
result = re.findall(r"a*", text)
```

 Copy

```
print(result) # ["aaaa", "", "", ""]
```

Ici, `a*` capture **tous les groupes** de "a" qui apparaissent 0 ou plusieurs fois.

3. Plus (+)

- Signification : Le plus `+` signifie **une ou plusieurs répétitions**.
- Exemple :

```
text = "hello aa bb aaa"  
result = re.findall(r"a+", text)  
print(result) # ["aa", "a", "aaa"]
```

 Copy

Ici, `a+` capture **toutes les occurrences** de "a" qui apparaissent une ou plusieurs fois.

4. Parenthèses ()

- Signification : Les **parenthèses** servent à capturer un **groupe** de caractères ou d'expressions.
- Exemple :

```
text = "nom: Pierre, âge: 30"  
result = re.findall(r"(\w+): (\w+)", text)  
print(result) # [('nom', 'Pierre'), ('âge', '30')]
```

 Copy

Ici, les parenthèses permettent de **capturer** "nom" et "Pierre", ainsi que "âge" et "30" comme des groupes distincts.

5. Crochets []

- Signification : Les **crochets** permettent de définir un **ensemble** de caractères. Cela signifie que **l'un des caractères** dans les crochets peut correspondre.
- Exemple :

```
text = "chat chapeau chair"  
result = re.findall(r"ch[aeiou]t", text)  
print(result) # ['chat']
```

 Copy

Ici, `[aeiou]` correspond à **n'importe quelle voyelle** entre `a` et `e` dans "chat".

6. Barre verticale ()

- Signification : La **barre verticale** est utilisée comme un **OU logique**.
- Exemple :

```
text = "chat chien"  
result = re.findall(r"chat|chien", text)  
print(result) # ['chat', 'chien']
```

 Copy

Ici, `chat|chien` cherche soit le mot "chat" soit le mot "chien" dans la chaîne.

7. Antislash ()

- Signification : Le **backslash** (antislash) est utilisé pour échapper des caractères spéciaux.
- Exemple :

```
text = "Un fichier avec un point. "
result = re.findall(r"\.", text)
print(result) # ['.]
```

 Copy

Ici, `\.` cherche le **caractère point** littéral, pas son rôle spécial dans regex.

4. Résumé avec quelques exemples :

Caractère	Signification	Exemple	Correspond à
.	N'importe quel caractère sauf saut de ligne	h.lllo	"hello"
*	0 ou plusieurs répétitions de ce qui précède	a*	"", "a", "aa", etc.
+	1 ou plusieurs répétitions de ce qui précède	a+	"a", "aa", "aaa"
()	Capturer un groupe	(\d+)	"123"
[]	Ensemble de caractères possibles	[aeiou]	"a", "e", "i", etc.
,	'		OU logique `chat
\	Échapper un caractère spécial	\.	"."

5. Exemple pratique avec des expressions régulières :

Imaginons que tu veuilles extraire les numéros de téléphone d'une chaîne, où le format est quelque chose comme "06-12345678".

Code :

```
import re
text = "Mon numéro est : 06-12345678"
result = re.findall(r"(\d{2})-(\d{8})", text)
print(result) # [('06', '12345678')]
```

 Copy

- Pattern `(\d{2})-(\d{8})` :
 - `\d{2}` correspond à **deux chiffres**.
 - `-` correspond à un **tiret littéral**.
 - `\d{8}` correspond à **huit chiffres**.

Le symbole `|` en expressions régulières représente une **alternance** ou un "ou" logique. Il permet de choisir entre plusieurs motifs possibles.

Par exemple, dans un motif `(\w+|\?|!)`, cela signifie que l'expression régulière correspond soit à un ou plusieurs caractères de mot (`\w+`), soit à un point d'interrogation (`?`) ou un point d'exclamation (`!`).

Cette logique est très utile pour faire des choix entre différentes options possibles dans une chaîne.

Analyser chaque motif proposé :

1. `r"(\w+|\?|!)"`

- **Signification :**

- `\w+` : correspond à une ou plusieurs lettres, chiffres ou underscore (caractères de mots).
- `\?` : correspond à un point d'interrogation.
- `!` : correspond à un point d'exclamation.

- **Explication :**

- Ce motif est assez simple. Il capte **les mots** (composés de lettres, chiffres ou underscores), mais aussi **les ponctuations "?" et "!"**.
- **Problème :** Cela ne retient pas "#1" comme un seul jeton, car `\w+` correspond à des mots mais ne capture pas les symboles spéciaux comme `#` dans un format spécifique.
- **Choix ou rejet :** **Rejeté** car il ne permet pas de conserver "#1" comme un seul jeton, ce qui est un critère important.

2. `r"(\w+|#\d|\?|!)"`

- **Signification :**

- `\w+` : correspond à une ou plusieurs lettres, chiffres ou underscore.
- `#\d` : correspond au caractère `#` suivi d'un chiffre.
- `\?` et `!` : correspondent respectivement aux caractères "?" et "!".

- **Explication :**

- Ce motif correspond aux **mot**s, aux **numéros précédés du caractère #** (comme `#1`), et à la ponctuation "?" et "!".
- Ce motif permet de conserver "#1" comme un seul jeton, ce qui est exactement ce que nous voulons.
- **Choix ou rejet :** **Choisi** car il permet de capturer les mots, les ponctuations, et aussi "#1" comme un seul jeton, respectant le besoin exprimé.

3. `r"(\#\d\w+\?!)"`

- **Signification :**

- `#\d` : correspond au caractère `#` suivi d'un chiffre.
- `\w+` : correspond à un ou plusieurs caractères de mots (lettres, chiffres, underscore).
- `?!` : correspond à un point d'interrogation suivi d'un point d'exclamation (littéralement les deux collés).
- **Explication :**
 - Ce motif correspond à un **nombre précédent du caractère #**, suivi d'un mot, puis de la ponctuation `?!`.
 - Ce motif ne capture pas les mots séparés comme "SOLDIER" ou "Found". Il capture en fait un format très spécifique comme `#1Found?!`, ce qui n'est pas ce que nous cherchons.
- **Choix ou rejet :** ❌ Rejeté car il ne permet pas de capturer correctement les mots ou la ponctuation indépendante.

4. `r"\s+"`

- **Signification :**
 - `\s+` : correspond à **un ou plusieurs espaces blancs** (espaces, tabulations, nouvelles lignes).
- **Explication :**
 - Ce motif correspond uniquement aux **espaces blancs**, et non aux mots ou à la ponctuation. Cela ne capture rien de ce qui est demandé dans la tâche.
- **Choix ou rejet :** ❌ Rejeté car il ne permet pas de capturer des tokens significatifs comme les mots ou la ponctuation.

Conclusion :

Le meilleur motif est donc `r"(\w+|#\d|\?|!)"`, car il permet de :

- Capturer des **mot**s.
- Capturer les **symboles de ponctuation** (`?` et `!`).
- Capturer "`#1`" comme un seul jeton.

C'est exactement ce que nous recherchons pour un tokenizer adapté.

Importance de l'ordre dans les expressions régulières

Oui, l'**ordre des motifs dans une expression régulière** est important et pris en compte lors de la correspondance.

Lorsque plusieurs options sont possibles dans une alternance (`|`), le moteur de recherche examine les motifs dans l'ordre dans lequel ils apparaissent, et il choisit le **premier qui correspond**.

Exemple par motif :

1. `r"(\w+|\?|!)"`

- **Ordre pris en compte :** Oui.
- **Explication :** Le moteur cherche d'abord à faire correspondre un ou plusieurs caractères de mots (`\w+`). Si cela échoue, il vérifie ensuite `?` ou `!`.
- **Exemple :**
 - `"SOLDIER"` → `\w+`
 - `"?"` → `?`

2. `r"(\w+|\#\d|\?|!)"`

- **Ordre pris en compte :** Oui.
- **Explication :** Le moteur teste d'abord les mots (`\w+`), ensuite `#\d`, puis la ponctuation.
- **Exemple :**
 - `"#1"` → `#\d`
 - `"SOLDIER"` → `\w+`

3. `r"(\#\d\w+\?!)"`

- **Ordre pris en compte :** Oui.
- **Explication :** L'ordre exact `#`, chiffre, mot, puis `?!` est requis. Toute variation échoue.
- **Exemple :**
 - `"#1Found?!"` → correspond
 - `"#1"` → ne correspond pas

4. `r"\s+"`

- **Ordre pris en compte :** N/A (pas d'alternance).
- **Explication :** Il ne contient qu'un seul motif.

🔁 Résumé :

Quand tu utilises `|` dans une regex :

- Le moteur lit de gauche à droite.
- Il choisit la première option qui matche.
- Donc, l'ordre influence directement le résultat de la correspondance.

L'instruction suivante en Python :

```
from nltk.tokenize import regexp_tokenize, TweetTokenizer
```

 Copy

importe deux **fonctions ou classes** différentes à partir du module `nltk.tokenize`, qui fait partie de la bibliothèque **NLTK** (Natural Language Toolkit), utilisée pour le **traitement du langage naturel (NLP)**.

◆ `regexp_tokenize`

- **C'est quoi ?**

C'est une fonction qui permet de **tokenizer** (découper en unités) un texte en utilisant une **expression régulière personnalisée**.

- **Utilité :**

Elle est très utile lorsque vous avez besoin d'un contrôle précis sur la manière de découper le texte. Par exemple, si vous voulez extraire des mots, des hashtags, des ponctuations, ou autre selon des règles spécifiques.

- **Exemple :**

```
from nltk.tokenize import regexp_tokenize
```

 Copy

```
text = "Hello #world! How are you? #2024"
tokens = regexp_tokenize(text, pattern=r"(\w+|#\d+|\?|!|)")
print(tokens)
# Sortie possible : ['Hello', '#2024', 'world', 'How', 'are', 'you', '?']
```

◆ `TweetTokenizer`

- **C'est quoi ?**

C'est une **classe spécialisée** dans le tokenization de **tweets** (ou de messages courts comme dans les réseaux sociaux).

- **Pourquoi est-elle utile ?**

Les tweets contiennent souvent des **hashtags** (#), des **mentions** (@), des **emojis**, des **abréviations**, etc. `TweetTokenizer` est conçu pour reconnaître ces éléments correctement, contrairement aux tokenizers classiques.

- **Exemple d'utilisation :**

```
from nltk.tokenize import TweetTokenizer
```

 Copy

```
tknzr = TweetTokenizer()
text = "Hello @user! #awesome 😊"
tokens = tknzr.tokenize(text)
print(tokens)
```

```
# Sortie possible : ['Hello', '@user', '!', '#awesome', '😊']
```

🔁 Différence entre les deux :

Fonction / Classe	Personnalisable	Gère les tweets / réseaux sociaux	Requiert une expression régulière
regexp_tokenize	<input checked="" type="checkbox"/> Oui	✗ Non (sauf si vous l'adaptez)	<input checked="" type="checkbox"/> Oui
TweetTokenizer	✗ Non	<input checked="" type="checkbox"/> Oui	✗ Non

Si tu veux tokenizer selon une règle que tu choisis → utilise `regexp_tokenize`.

Si tu veux tokenizer un texte type tweet avec hashtags, emojis, mentions → utilise `TweetTokenizer`.

Souhaites-tu un exemple comparatif des deux avec le même texte ?

❓ Pourquoi on écrit `([#@]\w+)` et pas `(#@\w+)` ?

L'expression suivante :

```
pattern = r"([#@]\w+)"
```

 Copy

est **correcte** pour extraire les hashtags `#mot` et les mentions `@mot`, tandis que :

```
pattern = r"(#@\\w+)"
```

 Copy

est **incorrecte** dans la plupart des cas.

🔍 Décryptage de `([#@]\w+)`

Élément	Signification
<code>[#@]</code>	Classe de caractères → correspond à soit <code>#</code> , soit <code>@</code> (un seul caractère)
<code>\w+</code>	Une ou plusieurs lettres, chiffres ou underscores (<code>[a-zA-Z0-9_]</code>)
<code>([#@]\w+)</code>	Capture donc des hashtags ou mentions classiques : <code>#Python</code> , <code>@nltk</code>

 C'est exactement ce qu'on veut pour du texte type Twitter.

✗ Pourquoi `(#@\\w+)` est incorrect

Élément	Signification
<code>#@</code>	Ce n'est pas une classe de caractères ! Il faut que le texte contienne exactement la séquence <code>#@</code> collée
<code>\w+</code>	Ensuite, un mot (comme <code>weird</code> , <code>user</code> , etc.)
<code>(#@\\w+)</code>	Capture des mots comme <code>#@user</code> — ce qui est très rare voire incorrect dans un vrai tweet

 Donc ce pattern ne trouvera rien d'utile sauf cas très spécifique.

📊 Exemple comparatif

```
import re
```

 Copy

```

text = "Check out #Python and follow @nltk for updates. Not #@weirdstuff"

# ✅ Pattern correct
print(re.findall(r"([#@]\w+)", text))
# Résultat : ['#Python', '@nltk']

# ❌ Pattern incorrect
print(re.findall(r"(@\w+)", text))
# Résultat : ['#@weirdstuff']

```

✓ Récapitulatif

Pattern	Description	Utilité pour NLP ?
([#@]\w+)	Capture un <code>#</code> ou <code>@</code> suivi d'un mot	✅ Oui
(@\w+)	Capture uniquement <code>@mot</code>	❌ Non

Explication complète de l'exercice avec TweetTokenizer de NLTK

Objectif de l'exercice

Tu travailles avec des **tweets** — un type de texte particulier qui contient souvent :

- des **hashtags** (`#nlp`)
- des **mentions** (`@datacamp`)
- des **émoticônes** (`:`, `<3`)
- des **ponctuations** (`!`, `?`)
- et parfois du **langage informel** (abréviations, fautes, etc.)

L'objectif est de **tokeniser** ces tweets de façon correcte.

Autrement dit, découper chaque tweet en **unités de sens** appelées *tokens* : mots, symboles, hashtags, etc.

Ce que tu dois faire

L'exercice te demande de :

1. Créer un **tokenizer** spécial pour les tweets grâce à `TweetTokenizer`.
2. Utiliser ce tokenizer pour **tokeniser chaque tweet** dans une liste de tweets (appelée `tweets`).
3. Afficher tous les tokens obtenus dans une nouvelle liste appelée `all_tokens`.

Solution

Voici le code que tu devais écrire :

```
from nltk.tokenize import regexp_tokenize
from nltk.tokenize import TweetTokenizer

# Étape 1 : Créer une instance de TweetTokenizer
tknzs = TweetTokenizer()

# Étape 2 : Utiliser une list comprehension pour tokeniser chaque tweet
all_tokens = [tknzs.tokenize(t) for t in tweets]

# Étape 3 : Afficher le résultat
print(all_tokens)
```

Exemple de tweets fournis (dans la variable `tweets`)

Imaginons qu'on ait ces 3 tweets :

```
tweets = [  
    "This is the best #nlp exercise ive found online! #python",  
    "#NLP is super fun! <3 #learning",  
    "Thanks @datacamp :) #nlp #python"  
]
```

 Copy

Résultat obtenu avec `TweetTokenizer`

```
[  
    ['This', 'is', 'the', 'best', '#nlp', 'exercise', 'ive', 'found', 'online',  
     '!', '#python'],  
    ['#NLP', 'is', 'super', 'fun', '!', '<3', '#learning'],  
    ['Thanks', '@datacamp', ':)', '#nlp', '#python']  
]
```

 Copy

Explication du résultat ligne par ligne :

Tweet 1 :

```
"This is the best #nlp exercise ive found online! #python"
```

 Tokenisé en :

```
['This', 'is', 'the', 'best', '#nlp', 'exercise', 'ive', 'found',  
 'online', '!', '#python']
```

 Copy

- Hashtags : `#nlp`, `#python` → bien reconnus
- Ponctuation : `!` → séparée comme un token
- Mots classiques aussi extraits : `This`, `exercise`, etc.

Tweet 2 :

```
"#NLP is super fun! <3 #learning"
```

 Tokenisé en :

```
['#NLP', 'is', 'super', 'fun', '!', '<3', '#learning']
```

 Copy

- Hashtags : #NLP , #learning 
- Émoticône : <3 
- Ponctuation : ! 

Tweet 3 :

```
"Thanks @datacamp :) #nlp #python"
```

 Tokenisé en :

```
['Thanks', '@datacamp', ':)', '#nlp', '#python']
```

 Copy

- Mention : @datacamp 
- Émoticône : :) 
- Hashtags : #nlp , #python 

Pourquoi utiliser `TweetTokenizer` ici ?

Parce que les tweets sont des textes non conventionnels :

- Le tokenizer standard ne reconnaîtrait pas les hashtags, mentions, émojis.
- `TweetTokenizer` est conçu pour comprendre le langage des réseaux sociaux.

Résumé

Élément	Reçu ?	Commentaire
Hashtag (#)		Identifié comme un mot spécial
Mention (@)		Reconnue comme un token
Émoticônes (:), <3)		Traitées comme un token entier
Ponctuation (!)		Séparée du mot
Mots courants		Standard, bien gérés

Cet exercice vous permet de vous exercer à la **tokenisation** de texte, mais cette fois-ci, avec des caractères **non-ASCII**, en incluant des **émoticônes** et des caractères spécifiques à la langue allemande. Le but de l'exercice est de comprendre et d'utiliser des fonctions de tokenisation avancées pour manipuler des textes complexes qui contiennent des symboles Unicode et des emoji.

Objectifs de l'exercice :

1. Tokeniser tous les mots dans une chaîne de texte en allemand, y compris les emoji.
2. Tokeniser uniquement les mots en majuscules.
3. Tokeniser uniquement les emoji, en utilisant des plages Unicode spécifiques.

Explication du code :

1. Tokenisation de tous les mots avec `word_tokenize()` :

```
all_words = word_tokenize(german_text)  
print(all_words)
```

 Copy

- `word_tokenize(german_text)` : La fonction `word_tokenize()` de la bibliothèque NLTK découpe un texte en mots (tokens). Cette fonction est utilisée pour séparer le texte `german_text` en **mots** et **emoji**, tout en tenant compte de la ponctuation.
- Dans le cas de la chaîne `german_text`, elle pourrait ressembler à quelque chose comme :

```
"Wann gehen wir Pizza essen ? 🍕 Und fährst du mit Über ? 🚖"
```

 Copy

La fonction `word_tokenize()` va couper ce texte en éléments distincts, comme les mots, les signes de ponctuation et même les emoji.

Sortie attendue :

```
['Wann', 'gehen', 'wir', 'Pizza', 'essen', '?', '🍕', 'Und', 'fährst', 'du', 'mit', 'Über', '?', '🚗']
```

 Copy

- Ici, vous voyez que les mots, les signes de ponctuation (comme `?`), et les emoji (`🍕`, `🚗`) sont tous séparés correctement.

2. Tokenisation des mots en majuscules :

```
capital_words = r"[A-ZÜ]\w+"  
print(regex_tokenizer(german_text, capital_words))
```

 Copy

- `capital_words = r"[A-ZÜ]\w+"` : Ce motif (regex) est conçu pour extraire uniquement les mots qui commencent par une lettre majuscule.
 - `[A-ZÜ]` : Cela capture les lettres majuscules de A à Z, ainsi que le caractère spécifique à l'allemand, `Ü`.
 - `\w+` : Cela capture le reste du mot (lettres, chiffres ou underscore).
- `regexp_tokenize(german_text, capital_words)` : Cette fonction `regexp_tokenize()` est utilisée pour découper le texte en fonction du motif regex. Elle extrait uniquement les mots qui commencent par une majuscule.

Sortie attendue :

```
['Wann', 'Pizza', 'Und', 'Über']
```

 Copy

- Seuls les mots qui commencent par une majuscule (y compris `Über`, qui est un mot allemand avec une majuscule au début) sont extraits ici. Les autres mots, comme `gehen` ou `essen`, qui ne commencent pas par une majuscule, sont ignorés.

3. Tokenisation des emoji :

```
emoji = "['\U0001F300-\U0001F5FF'|'\U0001F600-\U0001F64F'|'\U0001F680-\U0001F6FF'|'\u2600-\u26FF\u2700-\u27BF']"
print(regexp_tokenize(german_text, emoji))
```

 Copy

- `emoji` : Il s'agit d'une chaîne contenant des plages Unicode des emoji. Ces plages définissent des groupes de caractères Unicode pour différents types d'emoji :
 - `\U0001F300-\U0001F5FF` : Symboles divers.
 - `\U0001F600-\U0001F64F` : Visages souriants et émojis d'expressions faciales.
 - `\U0001F680-\U0001F6FF` : Symboles liés au transport.
 - `\u2600-\u26FF\u2700-\u27BF` : Symboles divers, comme des objets météorologiques ou des outils.
- Le but ici est de tokeniser uniquement les emoji dans `german_text`, en utilisant ces plages Unicode comme motif regex.

Sortie attendue :

```
['🍕', '🚕']
```

 Copy

- Les emoji `🍕` (pizza) et `🚕` (taxi) sont extraits du texte. Ces symboles sont capturés grâce à la plage Unicode définie dans le motif regex.

Résumé de la sortie et de ce qu'on cherche à atteindre :

Le code réalise trois tâches distinctes de tokenisation :

1. **Tokenisation complète** avec `word_tokenize()`, qui découpe tous les mots et les emoji.
 - **Sortie** : une liste de tous les mots et emoji du texte.
2. **Tokenisation des mots en majuscules** avec `regexp_tokenize()` en utilisant un motif regex qui capture les mots commençant par une majuscule (y compris des caractères comme `Ü`).
 - **Sortie** : une liste de mots qui commencent par une majuscule, par exemple, `Wann`, `Pizza`, `Und`, `Über`.
3. **Tokenisation des emoji** avec un motif regex qui capture des plages Unicode spécifiques aux emoji.
 - **Sortie** : une liste contenant les emoji extraits, par exemple,  et .

L'exercice vise à vous faire pratiquer la tokenisation avancée avec des **caractères non-ASCII**, en vous aidant à comprendre comment manipuler des textes contenant des caractères spéciaux comme des lettres majuscules et des emoji, tout en utilisant des expressions régulières pour capturer des modèles spécifiques.

Explication détaillée : "Charting word length with nltk"

Voici une explication détaillée de la vidéo "Charting word length with nltk", qui combine la tokenisation NLP avec la visualisation de données avec Matplotlib :

1. Introduction à la visualisation avec NLP (00:00 - 00:07)

"Hi everyone! In this video, we are going to learn about using charts with our NLP tools."

- L'objectif de la vidéo est d'apprendre à créer des graphiques avec les données extraites en traitement automatique du langage (NLP), en utilisant Python.

2. Démarrage avec Matplotlib (00:07 - 00:30)

"Matplotlib is a charting library..."

- Matplotlib est une bibliothèque de visualisation très utilisée en Python.
- Elle permet de créer :
 - des **histogrammes**
 - des **graphiques en barres**
 - des **courbes**
 - des **nuages de points**
 - et même des **graphiques 3D** ou des **animations**.
- Elle est souvent utilisée avec l'alias `plt` :

```
import matplotlib.pyplot as plt
```

 Copy

3. Tracer un histogramme simple (00:30 - 01:00)

"If we want to plot a basic histogram..."

- Un **histogramme** est un type de graphique qui montre la **distribution** de données.
- Exemple :

```
plt.hist([5, 5, 7, 7, 7])  
plt.show()
```

 Copy

- Ce code montre combien de fois chaque valeur apparaît.
- Ici :
 - `5` apparaît 2 fois

- 7 apparaît 3 fois

🔍 4. Histogramme généré (01:00 - 01:29)

"This is the chart that we generated..."

- Le graphique affiché reflète bien la **répartition** des nombres donnés.
- Matplotlib choisit automatiquement les "bacs" (bins) pour grouper les données.
- Le résultat est **fonctionnel**, mais peut être **amélioré visuellement** avec des options supplémentaires.

🧠 5. Combiner NLP et visualisation (01:29 - 02:27)

"We can then use skills we have learned..."

- On utilise maintenant les outils NLP (ex : `nltk`) pour **analyser du texte** :

```
from nltk.tokenize import word_tokenize
import matplotlib.pyplot as plt

sentence = "This is an example sentence."
words = word_tokenize(sentence)
lengths = [len(w) for w in words]
plt.hist(lengths)
plt.show()
```

 Copy

- Étapes :

1. **Tokenisation** : transformation du texte en mots.
2. **Calcul de la longueur de chaque mot** avec une **liste en compréhension** (`[len(w) for w in words]`).
3. Affichage d'un histogramme des **longueurs de mots**.

📊 6. Résultat : histogramme des longueurs de mots (02:27 - 02:48)

"Here is the generated histogram..."

- Le graphique montre que la majorité des mots font **4 lettres**.
- Cela est facile à observer ici car la phrase est courte.
- Mais si le texte est long (livre, pièce de théâtre), faire cela à la main serait **très fastidieux**
— le code **automatisé** cela efficacement.

✍ 7. À vous de jouer ! (02:48 - 02:54)

"Now it's your turn..."

- La vidéo vous encourage à **pratiquer** :
 - Tokenisation de texte avec NLTK
 - Création de graphiques avec Matplotlib
 - Visualisation des longueurs de mots, fréquences, etc.

Résumé final :

Étape	Description
1.	Importer les modules <code>nltk</code> et <code>matplotlib.pyplot</code>
2.	Tokeniser le texte avec <code>word_tokenize()</code>
3.	Utiliser une liste en compréhension pour obtenir les longueurs des mots
4.	Créer un histogramme avec <code>plt.hist()</code>
5.	Afficher le graphique avec <code>plt.show()</code>

 Souhaitez-tu que je te donne un exemple complet de code commenté à reproduire ? ``

Très bonne question! Parlons un peu de ce fameux `r""`, et pourquoi il est souvent utilisé (ou pas) avec les expressions régulières en Python.

🔍 Qu'est-ce que `r""` en Python ?

Le préfixe `r` devant une chaîne signifie "raw string" (chaîne brute). Cela indique à Python de ne pas interpréter les caractères spéciaux à l'intérieur de la chaîne.

Par exemple :

```
>>> print("C:\\Users\\Nom")
C:\\Users\\Nom

>>> print(r"C:\\Users\\Nom")
C:\\Users\\Nom
```

 Copy

Sans `r`, Python traite `\` comme un caractère d'échappement (ex : `\n`, `\t`, etc.).

📝 Pourquoi `r""` est utile avec les regex ?

Les expressions régulières utilisent beaucoup de `\` pour des choses comme :

- `\d` → chiffre
- `\s` → espace
- `\b` → bord de mot

Si on n'utilise pas `r""`, il faut doubler tous les `\` pour éviter les erreurs :

```
# Sans r"" → beaucoup d'échappements nécessaires
re.match("\b[A-Z]+\b", text)

# Avec r"" → plus lisible et plus propre
re.match(r"\b[A-Z]+\b", text)
```

 Copy

✅ Alors pourquoi ça marche ici sans `r""` ?

Ton pattern est :

```
pattern = "[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
```

 Copy

Regarde bien :

- Il n'y a pas de `\` qui pourrait être interprété comme un caractère spécial par Python.
- `\s` est la **seule** séquence avec un `\`, mais `\s` n'a pas besoin d'être échappé si tu l'écris dans une string normale. Python l'interprète sans problème.

⚠️ Mais attention : si tu avais écrit `\b`, `\n`, `\t`, etc., ça aurait causé des soucis.

💡 Conclusion

- Utiliser `r""` est **fortement recommandé** pour toutes les regex : c'est une **bonne pratique**.
- Même si ici ça marche sans `r""`, il vaut mieux écrire :

```
pattern = r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:"
```

 Copy

pour éviter tout bug futur si on ajoute des séquences comme `\b` ou `\n`.

Excellent question, vraiment ! On va décortiquer ça étape par étape : le rôle de chaque symbole dans les expressions régulières, comment fonctionne le pattern que tu as donné, et comment la capture fonctionne dans une liste. Let's go ↗

🔧 1. Que signifient [], { } , () en regex ?

Symbol	Rôle
[]	Classe de caractères : correspond à un seul caractère parmi ceux listés. Ex : [A-Z] → une lettre majuscule entre A et Z.
{ }	Quantificateur : indique combien de fois un motif doit apparaître. Ex : {2,} → au moins 2 fois.
()	Groupe capturant : permet de regrouper une portion de l'expression et de la capturer pour extraction. Ex : (\s)? capture un espace facultatif.

🧠 2. Décryptons ton pattern

```
pattern = r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:"
```

Copy

Voici le fonctionnement étape par étape :

Partie du pattern	Signification
[A-Z]{2,}	Au moins 2 lettres majuscules (ex : AB, NOTE, TEST).
(\s)?	Espace facultatif, capturé. Le ? signifie 0 ou 1 fois.
(#\d)?	Un # suivi d'un chiffre (\d) → aussi facultatif. Ex : #3.
([A-Z]{2,})?	Une deuxième séquence de majuscules, facultative.
:	Le tout doit se terminer par un deux-points.

✳️ Exemples de chaînes qui matchent ce pattern :

- "AB:"
- "NOTE #1:"
- "NOTE #2 TASK:"
- "TESTING:"

💡 3. Comment la regex fonctionne pour "capturer" les éléments dans une liste ?

Disons que tu as une liste de chaînes de texte :

```
lines = [  
    "NOTE #2 TASK: do something",  
    "AB: simple one",  
    "XYZ something",  
    "HELLO: just a test",  
    "MISSING #3:"  
]
```

 Copy

Et tu veux trouver uniquement celles qui matchent le pattern :

```
import re  
  
pattern = r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:"  
matches = [line for line in lines if re.match(pattern, line)]
```

 Copy

⌚ Ce qui se passe :

1. Python parcourt chaque élément (`line`) de la liste.
2. Il essaie d'appliquer le pattern au début de la ligne (car `re.match()` agit uniquement au début).
3. Si le pattern matche :
 - La ligne est conservée dans `matches`.
 - Les groupes capturés (avec `()` dans le pattern) peuvent être récupérés avec `.group()`.

📝 Pour capturer les parties spécifiques :

```
for line in matches:  
    m = re.match(pattern, line)  
    print("Texte:", line)  
    print("Groupe 1 (espace?) :", m.group(1))  
    print("Groupe 2 (#numéro?) :", m.group(2))  
    print("Groupe 3 (deuxième mot?) :", m.group(3))  
    print()
```

 Copy

📌 Résumé visuel

Expression	Rôle
[A-Z]	Lettre majuscule

 Copy

{2,}	Répéter au moins 2 fois
(...)	Capture (peut extraire ce groupe)
\s	Espace blanc
\d	Chiffre (0 à 9)
?	Rendre un élément facultatif
:	Deux-points littéral

🔍 Comprendre `re.match`, `re.search`, `re.findall`, `word_tokenize` et `RegexpTokenizer` étape par étape

📌 Objectif :

- Comprendre comment `re.match`, `re.search` et `re.findall` utilisent le pattern pour capturer.
- Puis voir comment `regex_tokenize()` (de `nltk`) et `word_tokenize()` fonctionnent pour capturer les tokens.

🧠 1. Est-ce que `re.match()` doit matcher tout le pattern au début ?

✓ OUI ! Mais attention à ce que ça veut dire :

- `re.match()` regarde uniquement à partir du début de la chaîne.
- Le pattern doit matcher dès la position 0, sinon il échoue.

⚠ Ça ne veut pas dire que toute la chaîne doit matcher. Juste que le motif doit apparaître immédiatement au début.

🔍 Exemple :

```
import re

text = "NOTE #2 TASK: do something"
pattern = r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:"
```

Copy

⌚ Ce que fait `re.match()` pas à pas :

1. Va au caractère `N`, position 0 ✓ (majuscule OK)
2. Trouve `[A-Z]{2,}` → `NOTE` ✓
3. Ensuite, il y a un espace → ça match `(\s)?` ✓
4. Puis `#2` → ça match `(#\d)?` ✓
5. Ensuite `TASK` → ça match `([A-Z]{2,})?` ✓
6. Il y a `:` → c'est bon ✓

→ Donc tout le motif est trouvé dès le début, donc `re.match()` réussit.



`re.search()`

: il scanne toute la chaîne

- Il parcourt toute la chaîne de gauche à droite.
- Dès qu'il trouve une sous-partie qui match le pattern, il s'arrête et retourne un match.

⌚ Différence clé :

- `re.match()` = position 0 uniquement
- `re.search()` = toute la chaîne

🔍 Exemple :

```
text = "Hello NOTE #2 TASK: let's go"
```

Copy

- `re.match()` → échoue ✗ (`Hello` ne match pas `[A-Z]{2,}`)
- `re.search()` → réussit ✓ car il trouve `"NOTE #2 TASK:"` plus loin



`re.findall()`

: retourne tous les matchs, selon les groupes

- Il scanne toute la chaîne comme `search()`.
- Il retourne une liste :
 - Si le pattern a des groupes () → retourne des tuples des groupes capturés
 - Sinon → retourne les chaînes complètes matchées

Exemple :

```
re.findall(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)  
# retourne : [(' ', '#2', 'TASK')]
```

Copy



Récap visuel clair

Fonction	Parcourt toute la chaîne ?	Retourne quoi ?	Condition
<code>re.match()</code>	✗ Non, uniquement au début	Objet <code>match</code> si début matché	Doit commencer par le motif

Fonction	Parcourt toute la chaîne ?	Retourne quoi ?	Condition
re.search()	<input checked="" type="checkbox"/> Oui	Premier <code>match</code> trouvé	Motif trouvé n'importe où
re.findall()	<input checked="" type="checkbox"/> Oui	Liste de chaînes ou tuples	Toutes les correspondances

2. Et les fonctions `regex_tokenize()` et `word_tokenize()` ?

💡 `nltk.word_tokenize()` (**tokeniseur de base**)

```
from nltk.tokenize import word_tokenize

text = "NOTE #2 TASK: do something!"
tokens = word_tokenize(text)
print(tokens)
```

⌚ Étapes de `word_tokenize()` :

1. Utilise une **regex prédéfinie** pour séparer :
 - Mots
 - Ponctuation
 - Nombres
2. Ne capture pas comme `re` → il **découpe**, pas capture.

📌 Retourne une **liste de tokens** :

```
['NOTE', '#', '2', 'TASK', ':', 'do', 'something', '!']
```

🔍 `nltk.RegexpTokenizer`

```
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r"[A-Z]{2,}|#\d|[a-z]+")
tokens = tokenizer.tokenize("NOTE #2 TASK: do something")
print(tokens)
```

⌚ Étapes de RegexpTokenizer :

1. Il parcourt toute la chaîne.
2. À chaque fois qu'il trouve une occurrence qui match le pattern, il l'ajoute à la liste.
3. Il ne capture que les correspondances → comme `re.findall()`.

📌 Résultat possible :

```
['NOTE', '#2', 'TASK', 'do', 'something']
```

 Copy

📌 Résumé général

Fonction	Type	Capture ?	Retour
<code>re.match()</code>	Regex	 Groupes	Objet match
<code>re.search()</code>	Regex	 Groupes	Objet match
<code>re.findall()</code>	Regex	 correspondances	Liste
<code>word_tokenize()</code>	NLP (NLTK)	 découpe	Liste de mots/ponctuation
<code>RegexpTokenizer()</code>	Regex + NLP (NLTK)	 découpe	Liste des match

📝 Démo pour voir les différences

```
from nltk.tokenize import word_tokenize, RegexpTokenizer
import re

text = "NOTE #2 TASK: do something"

# re.match
m = re.match(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)
print("match:", m.group() if m else None)

# re.search
s = re.search(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)
print("search:", s.group() if s else None)

# re.findall
f = re.findall(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)
print("findall:", f)

# word_tokenize
print("word_tokenize:", word_tokenize(text))

# RegexpTokenizer
```

 Copy

```
regex_tok = RegexpTokenizer(r"[A-Z]{2,}|\#\d|[a-z]+")
print("RegexpTokenizer:", regex_tok.tokenize(text))
```

Parfait, tu veux une **explication ultra détaillée**, non seulement étape par étape mais **sous-étape par sous-étape**, pour vraiment comprendre **comment fonctionnent** `re.match()`, `re.search()`, `re.findall()`, `word_tokenize()` et `RegexpTokenizer()`? Voici la version **Markdown** ultra détaillée avec des **sous-étapes**, comme une loupe sur chaque concept :

🔎 Analyse ultra détaillée des fonctions de tokenisation et des regex Copy en Python

🔥 Objectif :

- Comprendre **en profondeur** comment fonctionnent :
 - `re.match()`
 - `re.search()`
 - `re.findall()`
 - `word_tokenize()` (NLTK)
 - `RegexpTokenizer()` (NLTK)

🧐 1. Est-ce que `re.match()` doit matcher **tout le pattern au début** ?

✅ OUI, mais **voici ce que ça signifie vraiment** :

⚙ Fonctionnement interne de `re.match()` :

1. **Départ toujours à l'index 0** de la chaîne (`string[0]`)
2. Applique le **pattern regex complet** dès le départ
3. Si **la totalité du pattern matche depuis la position 0**, c'est un succès
4. Sinon, **aucune autre position ne sera testée**

🔎 Exemple détaillé :

```
```python
import re

text = "NOTE #2 TASK: do something"
pattern = r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:"
m = re.match(pattern, text)
```

## ⌚ Décryptage sous-pas à pas du pattern :

- `"[A-Z]{2,}"` :
  - `[A-Z]` = une majuscule
  - `{2,}` = au moins 2 lettres majuscules

- Match trouvé = "NOTE"
- `(\s)?` :
  - `\s` = un espace, tab, saut de ligne...
  - `?` = facultatif
  - Ici : espace entre "NOTE" et "#2" → match "
- `(#\d)?` :
  - `#` = caractère littéral
  - `\d` = un chiffre
  - `?` = entier facultatif
  - Match trouvé = "#2"
- `([A-Z]{2,})?` :
  - Encore un mot tout en majuscules
  - Match trouvé = "TASK"
- `:` = caractère littéral deux-points
  - Match trouvé = ":"

Tout le pattern est validé depuis l'index 0 → `re.match()` réussit

## 2. Fonctionnement de `re.search()` — comparaison et sous-étapes

### Principe de base :

- Balaye toute la chaîne
- Teste le pattern à chaque position
- S'arrête au premier match

### Exemple :

```
text = "Hello NOTE #2 TASK: let's go"
pattern = r"[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
```

Copy

### Déroulement interne (position par position) :

1. Index 0 : "H" → pas de majuscule répétée (échoue)
2. Index 1 : "e" → non majuscule → échoue
3. Index 6 : "N" → début potentiel
  - `[A-Z]{2,}` → NOTE
  - `\s` → espace

- #2 → ok ✓
- TASK → ok ✓
- : → ok ✓
- + Succès à partir de la position 6

`re.search()` réussit là où `re.match()` avait échoué



## 3. Fonctionnement de `re.findall()`

**But :** retourner toutes les correspondances dans une liste

- Scanne toute la chaîne (comme `search()`)
- Ne s'arrête pas au premier match
- Retourne :
  - soit tous les motifs complets
  - soit les groupes capturés

**Exemple avec groupes :**

```
f = re.findall(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", "NOTE #2 TASK:
NEXT TASK:")
print(f)
```

Copy

**Sous-détails :**

1. Match 1 : "NOTE #2 TASK:" → Groupes = (' ', '#2', 'TASK')
2. Match 2 : "NEXT TASK:" → Groupes = (None, None, 'TASK')

**Résultat :**

```
[(' ', '#2', 'TASK'), (None, None, 'TASK')]
```

Copy



## 4. Fonctionnement de `word_tokenize()` (NLTK)

**But :** découper un texte en mots, ponctuation, symboles

```
from nltk.tokenize import word_tokenize
```

Copy

```
text = "NOTE #2 TASK: do something!"
tokens = word_tokenize(text)
```

### ⌚ Sous-pas à pas :

1. Analyse "NOTE" → majuscule + lettres → ✓ mot
2. Analyse "#" → symbole → ✓ séparé
3. Analyse "2" → chiffre → ✓ séparé
4. Analyse "TASK" → majuscule + lettres → ✓ mot
5. Analyse ":" → ponctuation → ✓ séparée
6. Analyse "do" → mot → ✓
7. Analyse "something" → mot → ✓
8. Analyse "!" → ponctuation → ✓

### ✓ Résultat :

```
['NOTE', '#', '2', 'TASK', ':', 'do', 'something', '!']
```

 Copy

## 🔍 5. Fonctionnement de RegexpTokenizer (NLTK)

```
from nltk.tokenize import RegexpTokenizer

tokenizer = RegexpTokenizer(r"[A-Z]{2,}|#\d|[a-z]+")
tokens = tokenizer.tokenize("NOTE #2 TASK: do something")
```

### 📝 Sous-séquence d'analyse :

1. [A-Z]{2,} → "NOTE" ✓
2. \#\d → "#2" ✓
3. [A-Z]{2,} → "TASK" ✓
4. : → ignoré ✗
5. [a-z]+ → "do" ✓
6. [a-z]+ → "something" ✓

### ✓ Résultat :

```
['NOTE', '#2', 'TASK', 'do', 'something']
```

 Copy

## 💡 Tableau résumé des comportements

Fonction	Position	Type de retour	Capture ou découpe	Détail
re.match()	Début (index 0)	Objet match	🎯 Groupes capturés	Doit matcher au début
re.search()	Toute la chaîne	Objet match	🎯 Groupes capturés	1er match trouvé
re.findall()	Toute la chaîne	Liste (groupes)	✓ Toutes les captures	Multi-matches
word_tokenize()	NLP - règles internes	Liste	✗ Pas de capture	Sépare tout
RegexpTokenizer()	Regex perso	Liste	✓ Captures brutes	Regex filtrante

## 📝 Démo globale (code complet)

```
from nltk.tokenize import word_tokenize, RegexpTokenizer
import re

text = "NOTE #2 TASK: do something"

re.match
m = re.match(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)
print("match:", m.group() if m else None)

re.search
s = re.search(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)
print("search:", s.group() if s else None)

re.findall
f = re.findall(r"[A-Z]{2,}(\s)?(\#\d)?([A-Z]{2,})?:", text)
print("findall:", f)

word_tokenize
print("word_tokenize:", word_tokenize(text))

RegexpTokenizer
regex_tok = RegexpTokenizer(r"[A-Z]{2,}|\#\d|[a-z]+")
print("RegexpTokenizer:", regex_tok.tokenize(text))
```

## ❖ Tu veux aller encore plus loin ?

- Voir la différence entre groupes capturants et non-capturants ?
- Ajouter des regex avec lookaheads/lookbehinds ?
- Créer un notebook interactif avec visualisation ligne par ligne ?

Dis-moi et je te prépare ça sur-mesure 🧠💻

Très bonne question. Tu touches ici au cœur du fonctionnement des expressions régulières (regex) et de la manière dont elles analysent le texte.

On va tout détailler sous-step par sous-step, sans rien sauter. 

### Question :

Quand un pattern est composé de plusieurs éléments (par exemple `\d+[a-z]+`), est-ce que chaque élément (comme `\d+`, puis `[a-z]+`) parcourt tout le texte indépendamment, ou bien est-ce qu'ils travaillent ensemble caractère par caractère ?

### Réponse en une phrase :

Les éléments du pattern travaillent ensemble en une seule passe, gauche à droite, en suivant une logique de backtracking (retour arrière si un bout du motif échoue).

### Sous-step par sous-step : ce qui se passe

Prenons ce pattern :

```
pattern = r"\d+[a-z]+"
text = "123abc"
```

 Copy

#### Étape 0 — Compilation

Python compile le motif : `\d+` suivi de `[a-z]+` → ça signifie :

- `\d+` = 1 ou plusieurs chiffres
- `[a-z]+` = 1 ou plusieurs lettres minuscules

Le moteur regex va essayer de trouver une séquence dans le texte qui match ENTIEREMENT ce motif, pas morceau par morceau indépendant.

#### Étape 1 — Commencer au début du texte (`'1'`)

Il commence à la position 0 et essaie d'appliquer tout le motif `\d+[a-z]+` d'un coup :

- Est-ce que `1` est un chiffre ?  → on continue
- `2` → chiffre ? 
- `3` → chiffre ? 

→ jusque-là on a `\d+ = "123"`

Maintenant on continue avec `[a-z]+` :

- `a` → lettre ?
- `b` → lettre ?
- `c` → lettre ?

→ `[a-z]+ = "abc"`

### Résultat :

- Motif complet trouvé : `"123abc"`
- Il retourne **immédiatement** ce match
- Il ne continue pas à chercher plus loin (sauf si on appelle `findall()` ou on active un mode global)

## **⚠ Important : les éléments ne parcouruent PAS le texte séparément**

Ce n'est PAS : "`\d+` parcourt tout le texte, trouve tous les chiffres, PUIS `[a-z]+` parcourt tout le texte à part."

C'est plutôt : le pattern complet avance caractère par caractère, en testant chaque morceau du motif dans l'ordre, en essayant de coller à une portion du texte d'un seul tenant.

### Et s'il y a une erreur ? (Backtracking)

Imaginons :

```
text = "123A"
pattern = r"\d+[a-z]+"
```

 Copy

Le moteur essaie :

- `\d+ → "123"`
- `[a-z]+ → "A"`  → majuscule ≠ [a-z]

### Que fait-il ?

Il se dit : "Ok, j'ai accepté `"123"` pour `\d+`, mais peut-être que j'ai pris trop de chiffres."

Mais ici, on a aucune combinaison possible pour faire fonctionner `[a-z]+`, donc...

### Résultat :

Aucune correspondance. Retourne `None`.

## ✍ Autre exemple détaillé :

```
text = "12abc34def"
pattern = r"\d+[a-z]+"
```

 Copy

### Que se passe-t-il ?

1. Position 0 → 1 , 2  → \d+ = "12"
2. a , b , c  → [a-z]+ = "abc" → Motif trouvé : "12abc"

Il s'arrête là (si `re.search()`), ou continue (si `re.findall()`)

## 📌 Conclusion visuelle

```
Pattern : \d+ [a-z]+
Texte : 1 2 3 a b c

 ↑ Start
 \d+ = 123
 ↑ Next
 [a-z]+ = abc

→ Match : "123abc"
```

 Copy

## 🧠 Résumé final :

Question	Réponse
Est-ce que chaque élément du pattern parcourt le texte seul ?	✗ Non, ils travaillent <b>ensemble</b> dans une seule passe
Le pattern est-il évalué élément par élément en ordre ?	✓ Oui, de gauche à droite, chaque partie du motif est testée à la suite
Le moteur revient-il en arrière si ça ne marche pas ?	✓ Oui, c'est ce qu'on appelle le <b>backtracking</b>
Est-ce qu'il sort du test dès qu'il trouve un match ?	✓ Pour <code>search()</code> ou <code>match()</code> , oui. Pour <code>findall()</code> , il continue.

Si tu veux, je peux te faire une **simulation visuelle étape par étape** avec des dessins ou graphes pour illustrer ça aussi 

Tu veux ?

Dans l'exemple que tu as donné :

```
text = "12abc34def"
pattern = r"\d+[a-z]+"
```

 Copy

Le motif `\d+` (qui correspond à un ou plusieurs chiffres) commence à analyser le texte **dès le début**. Il commence à position 0 et trouve "12" comme correspondance pour `\d+`.

Ensuite, lorsque `\d+` a trouvé sa correspondance, **le moteur passe directement à la position où il s'est arrêté**, c'est-à-dire **juste après "12"**. C'est là qu'il commence à tester `[a-z]+` (qui correspond à un ou plusieurs caractères minuscules) sur le texte restant.

## En d'autres termes :

- Le motif `\d+` commence à partir du début du texte, et une fois qu'il trouve une correspondance, il **avance dans le texte à la position suivante**.
- Ensuite, le motif suivant `[a-z]+` commence à être testé à partir de cette position, c'est-à-dire juste après les chiffres que `\d+` a matchés.

## Illustration détaillée :

1. Le moteur commence à position 0 et teste `\d+` :

- 1 (chiffre) →
- 2 (chiffre) →
- → Donc, `\d+` matché "12".

2. Une fois que `\d+` a matché "12", le moteur passe à la position **après ce match**, c'est-à-dire à la position où il trouve "a".

3. Maintenant, le moteur teste `[a-z]+` :

- a (lettre minuscule) →
- b (lettre minuscule) →
- c (lettre minuscule) →
- → Donc, `[a-z]+` matché "abc".

## Résultat :

Le moteur **combine** les deux matchs et retourne le premier match complet qu'il trouve : "12abc".

Ainsi, pour répondre à ta question, `[a-z]+` **commence là où \d+ s'est arrêté**, c'est-à-dire juste après les chiffres trouvés.

Si tu as d'autres questions ou si tu veux un autre exemple, n'hésite pas ! ``

Voici un exemple détaillé d'utilisation du **backtracking** avec des expressions régulières et une explication étape par étape. Nous allons utiliser un pattern qui échoue à un certain point, puis utilise le backtracking pour revenir en arrière et essayer une autre solution.

## Exemple détaillé : Backtracking

Imaginons que nous avons le texte suivant et un motif qui cherche une séquence de chiffres suivis de lettres minuscules :

```
import re
text = "12345abc"
pattern = r"\d{3}[a-z]{4}"
```

 Copy

Nous avons le motif `\d{3}[a-z]{4}`, ce qui signifie :

- `\d{3}` : exactement trois chiffres.
- `[a-z]{4}` : exactement quatre lettres minuscules.

### Objectif :

Nous voulons que le moteur regex trouve une séquence de **trois chiffres suivis de quatre lettres minuscules** dans le texte `"12345abc"`.

### Explication du backtracking :

Voici comment le moteur regex va procéder, étape par étape, avec **backtracking**.

#### Étape 0 — Compilation et début du test

Le moteur commence par analyser le texte `"12345abc"` à partir de la position 0 et tente de faire correspondre le motif complet `\d{3}[a-z]{4}`.

- `\d{3}` : il cherche d'abord trois chiffres consécutifs. La première séquence qu'il trouve est `"123"`.

Donc, `\d{3}` matché par `"123"`.

Le moteur avance ensuite à la position 3 (juste après `"123"`).

#### Étape 1 — Tester `[a-z]{4}` à partir de la position 3

À partir de la position 3, le moteur essaie de faire correspondre `[a-z]{4}`, c'est-à-dire quatre lettres minuscules consécutives.

Il trouve `"a"`, puis `"b"`, puis `"c"`, mais il reste encore un dernier caractère à tester pour atteindre les 4 lettres requises.

À la position 7, il essaie "5", mais "5" n'est pas une lettre minuscule, donc l'échec est immédiat pour [a-z]{4}.

⌚ Backtracking : Le moteur doit maintenant revenir en arrière pour essayer une autre possibilité.

#### □ Étape 2 — Le moteur essaie avec moins de chiffres pour \d{3}

Le moteur va revenir en arrière, car l'essai précédent a échoué. Il va essayer de réduire la correspondance de \d{3} à \d{2}.

- \d{2} peut être "12" (les deux premiers chiffres de "12345abc").

Il avance ensuite à la position 2, c'est-à-dire juste après "12".

#### □ Étape 3 — Tester [a-z]{4} à partir de la position 2

Maintenant, il tente de tester [a-z]{4} à partir de la position 2. Il trouve "3", "4", "5", mais il reste encore un dernier caractère à tester pour avoir 4 lettres.

Mais cela échoue également, car "5" n'est pas une lettre minuscule.

⌚ Backtracking : Le moteur revient à nouveau en arrière pour essayer une autre solution.

#### □ Étape 4 — Le moteur essaie encore moins de chiffres pour \d{3}

Le moteur essaie maintenant une correspondance avec seulement \d{1}.

- \d{1} matché par "1". Il avance donc à la position 1.

#### □ Étape 5 — Tester [a-z]{4} à partir de la position 1

Le moteur va maintenant tester [a-z]{4} en partant de la position 1.

- Il trouve "2", "3", "4", "5", mais échoue à atteindre la fin des caractères demandés pour la correspondance avec [a-z]{4}.

😊 Résultat : Aucun match trouvé.

## Conclusion du processus :

Le moteur a utilisé le **backtracking** pour revenir en arrière à chaque fois qu'il a échoué à faire correspondre le motif. Mais il n'a pas réussi à trouver de correspondance dans ce cas précis, car le motif `\d{3}[a-z]{4}` n'a pas pu correspondre entièrement à quoi que ce soit dans le texte "12345abc".

Voici la réponse en markdown avec un exemple de backtracking détaillé.

```
Exemple détaillé de Backtracking avec Expressions Régulières
```

 Copy

Prenons l'exemple suivant :

```
```python
import re

text = "12345abc"
pattern = r"\d{3}[a-z]{4}"
```

Le motif `\d{3}[a-z]{4}` signifie :

- `\d{3}` : exactement trois chiffres.
- `[a-z]{4}` : exactement quatre lettres minuscules.

Explication du Backtracking :

Voici comment le moteur regex va procéder, étape par étape, avec **backtracking**.

Étape 0 — Compilation et début du test

Le moteur commence par analyser le texte "12345abc" à partir de la position 0 et tente de faire correspondre le motif complet `\d{3}[a-z]{4}`.

1. `\d{3}` matché par "123".
2. Le moteur avance à la position 3 (juste après "123").

Étape 1 — Tester `[a-z]{4}` à partir de la position 3

À partir de la position 3, le moteur essaie de faire correspondre `[a-z]{4}`.

1. Il trouve "a", "b", puis "c", mais il n'atteint pas quatre lettres. Il échoue à "5", qui n'est pas une lettre minuscule.
2. **Backtracking** : Le moteur revient en arrière.

Étape 2 — Le moteur essaie avec moins de chiffres pour `\d{3}`

Le moteur va essayer une correspondance avec `\d{2}`.

1. Il match "12" et avance à la position 2.
2. À partir de cette position, il teste `[a-z]{4}` et échoue à "5" car ce n'est pas une lettre minuscule.

□ Étape 3 — Le moteur essaie encore moins de chiffres pour `\d{3}`

Le moteur essaie maintenant une correspondance avec `\d{1}`.

1. Il match "1" et avance à la position 1.
2. Il tente `[a-z]{4}` mais échoue encore à la position 5.

😊 Résultat : Aucun match trouvé

Le moteur n'a pas trouvé de correspondance après plusieurs essais de backtracking.

Ce processus montre comment le moteur regex utilise le **backtracking** pour revenir en arrière et tester différentes possibilités jusqu'à ce qu'il trouve un match ou qu'il échoue définitivement.

Exemple de Backtracking réussi

Voici un exemple où le moteur regex utilise le **backtracking** pour réussir à trouver des correspondances après plusieurs tentatives.

Exemple de texte et de motif

```
import re

text = "123abc456def"
pattern = r"\d{3}[a-z]{3}"
```

 Copy

Le motif `\d{3}[a-z]{3}` signifie :

- `\d{3}` : exactement trois chiffres.
- `[a-z]{3}` : exactement trois lettres minuscules.

Explication du Backtracking :

Étape 0 — Compilation et début du test

Le moteur commence à la position 0 et teste `\d{3}`. Il trouve "123" comme correspondance pour `\d{3}`.

`\d{3}` matché par "123".

Il avance donc à la position 3, juste après "123".

Étape 1 — Tester `[a-z]{3}` à partir de la position 3

À la position 3, il trouve "a", puis "b", puis "c", ce qui correspond à `[a-z]{3}`.

`[a-z]{3}` matché par "abc".

Résultat partiel :

Le moteur a trouvé la première correspondance "123abc".

Étape 2 — Backtracking : Revenir à la position 3

Le moteur revient maintenant à la position 3 pour tester d'autres possibilités.

Étape 3 — Tester "456" pour `\d{3}`

À la position 3, il tente maintenant de matcher "456" pour `\d{3}`.

Étape 4 — Tester `[a-z]{3}` à partir de la position 6

Il trouve "d", puis "e", puis "f", ce qui correspond à `[a-z]{3}`.

`[a-z]{3}` matché par "def".

Résultat final :

Les deux correspondances trouvées sont "123abc" et "456def" grâce au backtracking.

Bien sûr ! Voici une version en **Markdown** avec des explications **pas à pas (step by step)** de `regex_tokenize()` et `word_tokenize()` avec des **exemples bien détaillés** pour bien comprendre leur comportement.

```
# Comparaison détaillée : `regex_tokenize()` vs `word_tokenize()` Copy
```

En traitement du langage naturel (NLP), ****tokenizer**** signifie "découper une phrase en unités élémentaires appelées **tokens** (mots, ponctuations, etc.)".

Nous allons comparer ****`regex_tokenize()`**** et ****`word_tokenize()`**** en expliquant, étape par étape, ****comment**** chaque fonction découpe un texte.

```
## 🖊 1. `regex_tokenize()` – Personnalisable avec des motifs
```

```
### 🔍 Description :  
`regex_tokenize()` (souvent `re.findall()` en pratique) permet d'utiliser une  
**expression régulière (regex)** pour définir comment découper le texte.
```

```
### 📄 Exemple de texte :
```

```
```python  
text = "Hello, world! It's 2025. AI is amazing."
```

## 💡 Objectif :

Découper ce texte en mots uniquement (sans ponctuation).

## 🧠 Étapes :

```
import re Copy

tokens = re.findall(r'\b\w+\b', text)
print(tokens)
```

### ◻ Étape 0 — Analyse du motif :

- `\b` : limite de mot (début ou fin)
- `\w+` : une ou plusieurs lettres, chiffres ou underscore
- `\b` : fin du mot

Ce motif va trouver tous les "mots" sans ponctuation, car il ignore les caractères non alphanumériques.

### Résultat :

```
['Hello', 'world', 'It', 's', '2025', 'AI', 'is', 'amazing'] 
```

 Remarquez que "It's" est découpé en deux tokens : "It" et "s" car l'apostrophe n'est pas incluse dans `\w`.

## 2. `word_tokenize()` — Basé sur un modèle linguistique

### Description :

`word_tokenize()` est fourni par **NLTK**. Il utilise des règles linguistiques plus intelligentes pour gérer la ponctuation, les contractions, etc.

### Exemple identique :

```
from nltk.tokenize import word_tokenize

text = "Hello, world! It's 2025. AI is amazing."
tokens = word_tokenize(text)
print(tokens) 
```

### Étapes :

`word_tokenize` utilise le **Punkt Tokenizer** :

- Étape 0 — Charger les règles pré-apprises (ponctuation, abréviations, contractions).
- Étape 1 — Trouver les séparateurs : espaces, ponctuation, etc.
- Étape 2 — Gérer les **contractions** comme "It's" ou "I'm" intelligemment.

### Résultat :

```
['Hello', ',', 'world', '!', 'It', "'", 's"', '2025', '.', 'AI', 'is',
'amazing', '.'] 
```

🔍 Contrairement à `regex_tokenize`, ici on conserve la ponctuation **comme des tokens séparés**, et "It's" devient "It" et "'s".

## 📊 Comparaison visuelle

Fonction	Conserve ponctuation ?	Gère contractions ?	Personnalisable ?	Exemple "It's"
<code>re.findall()</code>	✗ Non	✗ Non	✓ Oui (regex)	"It", "s"
<code>word_tokenize()</code>	✓ Oui	✓ Oui	✗ Non (règles fixes)	"It", "'s"

## 🎯 Conclusion

- ✓ Utilisez `regex_tokenize()` quand vous voulez un contrôle **personnalisé** via des expressions régulières.
- ✓ Utilisez `word_tokenize()` quand vous voulez un découpage **linguistique** intelligent, notamment pour l'analyse de texte naturel.

## 🔄 Bonus : Créez votre propre tokenizer avec regex

Si vous voulez par exemple extraire **uniquement les mots**, en **ignorant les chiffres**, vous pouvez modifier votre regex :

```
Extraire uniquement les mots alphabétiques (sans chiffres)
re.findall(r'\b[a-zA-Z]+\b', text)
```

Copy

## 🧪 Tester vous-même

```
text = "Dr. John isn't here. It's 9:00 a.m. already!"
print(re.findall(r'\b\w+\b', text)) # regex_tokenize
print(word_tokenize(text)) # word_tokenize
```

Copy

# 🧪 Charting Practice – Solution Complète et Expliquée

## 📚 Objectif :

Afficher un histogramme du **nombre de mots par ligne** dans le script de *Holy Grail*, en :

1. Nettoyant les noms des personnages.
2. Tokenisant les lignes.
3. Comptant les mots de chaque ligne.
4. Traçant l'histogramme.

## ✓ Solution complète :

```
import re
from nltk.tokenize import regexp_tokenize
import matplotlib.pyplot as plt

1. Split the script into lines: lines
lines = holy_grail.split('\n')

2. Replace all script lines for speaker (e.g., 'ARTHUR:', 'SOLDIER #1:')
pattern = r"[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:"
lines = [re.sub(pattern, '', l) for l in lines]

3. Tokenize each line: tokenized_lines
tokenized_lines = [regexp_tokenize(s, pattern=r'\w+') for s in lines]

4. Make a frequency list of lengths: line_num_words
line_num_words = [len(t_line) for t_line in tokenized_lines]

5. Plot a histogram of the line lengths
plt.hist(line_num_words, bins=30, color='skyblue', edgecolor='black')

6. Show the plot
plt.title("Distribution du nombre de mots par ligne")
plt.xlabel("Nombre de mots")
plt.ylabel("Fréquence")
plt.show()
```

## Explication ligne par ligne :

```
```python
```

```
lines = holy_grail.split("\n")
```

 Copy

- ◆ **But** : séparer le texte en lignes individuelles.
- ◆ Exemple :

```
```python
```

```
holy_grail = "ARTHUR: Hello.\nSOLDIER #1: Who goes there?"
lines = ["ARTHUR: Hello.", "SOLDIER #1: Who goes there?"]
```

```
```python
```

```
pattern = r"[A-Z]{2,}(\s)?(#\d)?([A-Z]{2,})?:" lines = [re.sub(pattern, "", l) for l in lines]
```

 Copy

- ◆ **But** : supprimer les noms des personnages et ponctuation (comme `ARTHUR:` ou `SOLDIER #1:`).
- ◆ `re.sub()` remplace le motif par `''`.
- ◆ Exemple :

```
```python
```

```
"ARTHUR: Hello." → " Hello."
"SOLDIER #1: Who goes there?" → " Who goes there?"
```

```
```python
```

```
tokenized_lines = [regexp_tokenize(s, pattern=r'\w+') for s in lines]
```

 Copy

- ◆ **But** : découper chaque ligne en **mots seulement**, en ignorant la ponctuation.
- ◆ `\w+` : mots = lettres, chiffres, underscore.
- ◆ Exemple :

```
```python
```

```
" Who goes there?" → ['Who', 'goes', 'there']
```

### ```python

```
line_num_words = [len(t_line) for t_line in tokenized_lines]
```

 Copy

- ◆ \*\*But\*\* : calculer le \*\*nombre de mots par ligne\*\*.
- ◆ On utilise `len()` sur chaque ligne tokenisée.
- ◆ Exemple :

```
```python  
['Who', 'goes', 'there'] → 3  
['Hello'] → 1
```

```python

```
plt.hist(line_num_words, bins=30, color='skyblue', edgecolor='black')
```

 Copy

- ◆ **But** : dessiner un **histogramme** du nombre de mots par ligne.
- ◆ `bins=30` : on divise la plage de valeurs en 30 colonnes.
- ◆ `color` et `edgecolor` ajoutent un style sympa.

```
### ```python  
plt.title("Distribution du nombre de mots par ligne")  
plt.xlabel("Nombre de mots")  
plt.ylabel("Fréquence")  
plt.show()
```

- ◆ Affiche le graphe avec des titres.

Exemple de sortie (selon les données) :

Un histogramme montrant que la plupart des lignes ont environ 3 à 7 mots, avec quelques lignes très longues.



Word Counts with Bag-of-Words (BOW)

🎬 1. Introduction

⌚ 00:00 - 00:08

Bienvenue dans le **chapitre 2** !

Nous allons commencer avec le comptage de mots en utilisant l'approche **Bag-of-Words**.

💻 2. Qu'est-ce que le Bag-of-Words ?

⌚ 00:08 - 00:35

Le **Bag-of-Words (BOW)** est une méthode **simple** pour identifier les thèmes d'un texte.

Elle suit deux étapes :

1. **Tokenisation** : découper le texte en mots (ou *tokens*).
2. **Comptage des mots** : compter la fréquence de chaque mot.

◆ *Hypothèse* : plus un mot apparaît souvent, plus il est **important** pour le contenu.



📦 3. Exemple simple de Bag-of-Words

⌚ 00:35 - 01:17

Texte brut :

The cat **is in** the box. The cat likes the box.

Copy

◆ **Étape 1 : Tokenisation (sans ponctuation)**

```
['The', 'cat', 'is', 'in', 'the', 'box', 'The', 'cat', 'likes', 'the',  
 'box'] Copy
```

◆ **Étape 2 : Compte des mots (sans prétraitement)**

```
{  
    'The': 2, Copy
```

```
'cat': 2,  
'is': 1,  
'in': 1,  
'the': 3,  
'box': 2,  
'likes': 1  
}
```

✖ Problème :

- "The" et "the" sont comptés séparément.

🔧 Solution :

- Appliquer un **prétraitement** : tout mettre en **minuscule**.

```
['the', 'cat', 'is', 'in', 'the', 'box', 'the', 'cat', 'likes', 'the',  
 'box']  
Copy
```

Et maintenant :

```
{  
    'the': 4,  
    'cat': 2,  
    'is': 1,  
    'in': 1,  
    'box': 2,  
    'likes': 1  
}  
Copy
```

🐍 4. Implémentation en Python

⌚ 01:17 - 02:18

Voici comment construire un **Bag-of-Words** avec **NLTK** et **Counter** de Python :

```
from nltk.tokenize import word_tokenize  
from collections import Counter  
  
# Exemple de texte  
text = "The cat is in the box. The cat likes the box."  
  
# Prétraitement : mise en minuscule  
text = text.lower()  
Copy
```

```
# Tokenisation
tokens = word_tokenize(text)

# Bag-of-Words avec Counter
bow = Counter(tokens)

# Afficher le résultat
print(bow)

# Obtenir les 2 mots les plus fréquents
print(bow.most_common(2))
```

📋 Résultat :

```
Counter({'the': 4, 'cat': 2, 'box': 2, 'is': 1, 'in': 1, 'likes': 1}) ⏺ Copy
[('the', 4), ('cat', 2)]
```

📌 Remarques :

- `Counter` fonctionne comme un dictionnaire.
- `.most_common(n)` retourne les `n` mots les plus fréquents.
- Les mots avec la même fréquence ne sont pas forcément triés entre eux.

✍ 5. À vous de jouer !

⌚ 02:18 - 02:25

Tu connais maintenant les bases du **Bag-of-Words** !

Tu peux commencer à l'utiliser pour explorer le contenu de n'importe quel texte avec Python.

🧠 Résumé des étapes clés :

Étape	Action	Code clé
1	Nettoyage du texte	<code>text.lower()</code>
2	Tokenisation	<code>word_tokenize(text)</code>
3	Bag-of-Words	<code>Counter(tokens)</code>
4	Top mots	<code>bow.most_common(n)</code>

Construire un compteur avec le modèle bag-of-words (sac de mots)

Dans cet exercice, on va construire un **compteur de mots** basé sur l'article Wikipedia stocké dans la variable `article`. L'objectif est de créer une **représentation simple** du texte sous forme de **bag-of-words**, c'est-à-dire un dictionnaire où chaque mot est associé à son nombre d'occurrences dans le texte.

Code Complet avec Explication

```
# Import Counter
from collections import Counter
```

 Copy

Explication :

On importe la classe `Counter` depuis le module `collections`. `Counter` permet de compter facilement les occurrences d'éléments dans une liste.

```
# Tokenize the article: tokens
tokens = word_tokenize(article)
print("tokens :\n", tokens)
```

 Copy

Explication :

On utilise la fonction `word_tokenize()` (déjà importée) pour diviser l'article en une liste de `tokens` (mots, ponctuations, symboles, etc.).

Exemple de sortie partielle possible :

```
tokens :
['Debugging', 'is', 'the', 'process', 'of', 'finding', 'and', 'resolving',
'bugs', ',', 'or', 'defects', '...', '.']
```

 Copy

```
# Convert the tokens into lowercase: lower_tokens
lower_tokens = [t.lower() for t in tokens]
print("lower_tokens :\n", lower_tokens)
```

 Copy

Explication :

On convertit tous les tokens en **minuscules** pour ne pas distinguer par exemple `The` et `the`. Cela améliore la cohérence du comptage.

Exemple de sortie :

```
lower_tokens :  Copy
['debugging', 'is', 'the', 'process', 'of', 'finding', 'and', 'resolving',
'bugs', ',', 'or', 'defects', ...]
```

```
# Create a Counter with the lowercase tokens: bow_simple  Copy
bow_simple = Counter(lower_tokens)
```

Explication :

On crée un objet `Counter` à partir de `lower_tokens`. Chaque mot sera associé à son **nombre d'apparitions** dans le texte.

```
# Print the 10 most common tokens  Copy
print("bow_simple", bow_simple.most_common())
```

Explication :

On affiche les mots les plus fréquents du texte grâce à la méthode `.most_common()`, qui retourne une liste triée par fréquence décroissante.

Résultat attendu

Voici un aperçu des 10 tokens les plus fréquents :

```
[(',', 151), ('the', 150), ('.', 89), ('of', 81), ('"', 66),
('to', 63), ('a', 60), ('``', 47), ('in', 44), ('and', 41)]  Copy
```

Analyse du résultat :

- **Ponctuation** comme `,`, `.` ou `''` est très fréquente, ce qui est courant dans un texte non nettoyé.
- **Mots outils** comme `the`, `of`, `to`, `a`, `in`, `and` sont aussi très présents. Ce sont des **stop words** en NLP (souvent ignorés dans des analyses plus poussées).
- Le mot **debugging** apparaît 40 fois, ce qui indique probablement que **le sujet de l'article est le débogage** (debugging).

Conclusion

Ce code nous a permis de :

1. **Tokeniser** un article brut de Wikipedia.
2. **Nettoyer partiellement** (minuscule uniquement).
3. Crée une **fréquence de mots** via un bag-of-words.
4. Identifier les **mots dominants** dans l'article.

👉 Ce genre de bag-of-words est souvent utilisé comme **base** pour l'analyse de texte, la classification, ou encore la construction de modèles de machine learning.

Simple Text Preprocessing (Prétraitement de texte simple)

1. Introduction

 00:00 - 00:06

Dans cette vidéo, nous allons couvrir quelques techniques simples de prétraitement de texte.

2. Pourquoi prétraiter ?

 00:06 - 01:06

Le prétraitement de texte permet d'améliorer la qualité des données d'entrée en machine learning ou pour d'autres méthodes statistiques.

Par exemple :

- Lors des précédents exercices, vous avez utilisé des techniques simples comme la **tokenisation** pour créer un modèle **Bag of Words**.
- Des étapes comme **mettre tout en minuscules** (`lowercasing`) peuvent légèrement améliorer les performances.

Autres techniques classiques :

- **Lemmatisation** : réduire les mots à leur forme canonique.
- **Stemming** : couper les mots à leur racine (souvent plus agressif que la lemmatisation).
- **Suppression des stopwords** : mots fréquents qui portent peu de sens (comme "et", "le", "de").
- **Suppression de la ponctuation** et des caractères non désirés.

 Chaque tâche et chaque modèle peut réagir différemment au prétraitement. Il est donc utile de tester plusieurs approches pour trouver celle qui convient le mieux.

3. Exemple de prétraitement

 01:06 - 01:31

Prenons un exemple simple :

Texte d'entrée :

Cats are chasing the mice. Dogs are barking loudly.

 Copy

Sortie attendue après prétraitement :

```
['cat', 'chase', 'mouse', 'dog', 'bark', 'loud']
```

 Copy

 On remarque que :

- Les mots sont en **minuscules**.
- Les **stopwords** ont été supprimés.
- Les mots au pluriel sont devenus **singuliers** (lemmatisation).

4. Prétraitement de texte avec Python

 01:31 - 02:52

Voici un exemple en **Python** utilisant NLTK :

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from collections import Counter

text = "The cat is sitting in the box. The cat likes the box."

# Tokenisation + passage en minuscules
tokens = [word for word in word_tokenize(text.lower()) if word.isalpha()]

# Suppression des stopwords
clean_tokens = [word for word in tokens if word not in
stopwords.words('english')]

# Fréquence des mots
word_freq = Counter(clean_tokens)
print(word_freq.most_common(2)) # [('cat', 2), ('box', 2)]
```

 Copy

 Étapes :

1. Texte mis en **minuscules**.
2. **Tokenisation** avec `word_tokenize`.
3. Filtrage avec `isalpha()` pour ne garder que les **mots alphabétiques**.
4. Suppression des **stopwords** via NLTK.
5. Crédation d'un **compteur de fréquence** avec `Counter`.

 Résultat : Les mots "the" ont été supprimés, et les plus fréquents sont "cat" et "box" — bien plus informatifs.

5. À vous de jouer !

 02:52 - 02:59

Vous pouvez maintenant vous entraîner à **prétraiter vos propres textes** !

 Essayez différentes combinaisons de techniques et observez leur impact sur vos modèles.

Syntaxe de la list comprehension

La **list comprehension** est une façon concise et élégante de créer une liste en Python.

Syntaxe générale :

```
[expression for element in iterable if condition]
```

 Copy

- **expression** : ce qu'on veut mettre dans la nouvelle liste (souvent le même `element`, mais pas toujours).
- **element** : chaque élément pris de `iterable`.
- **condition (facultative)** : filtre pour inclure seulement certains éléments.

Décryptage du code

```
# Ligne 1 : Tokenisation + passage en minuscules
```

 Copy

```
tokens = [word for word in word_tokenize(text.lower()) if word.isalpha()]
```

Étape par étape :

1. `text.lower()` :

- Convertit tout le texte en **minuscules**.
- Exemple : "The Cat" devient "the cat".

2. `word_tokenize(...)` :

- Coupe le texte en **mots** ou **tokens**.
- Exemple : "the cat" devient ["the", "cat"] .

3. `for word in ...` :

- On boucle sur **chaque mot** du texte tokenisé.

4. `if word.isalpha()` :

- On **garde seulement les mots composés uniquement de lettres** (pas de ponctuation, ni de chiffres).
- "cat" est gardé, "!" ou "123" sont ignorés.

 Résultat : `tokens` est une liste de mots en minuscules sans ponctuation ni chiffres.

```
# Ligne 2 : Suppression des stopwords
```

 Copy

```
clean_tokens = [word for word in tokens if word not in
```

```
stopwords.words('english')]
```

Étape par étape :

1. `for word in tokens :`
 - On parcourt chaque mot de la liste `tokens`.
2. `if word not in stopwords.words('english') :`
 - On filtre les **stopwords** (mots très courants qui n'apportent pas beaucoup d'information).
 - Exemples de stopwords : `["the", "is", "in", "and", "a", ...]`.
3. `stopwords.words('english')` :
 - Appelle une **liste de stopwords** fournie par la bibliothèque **NLTK**.

👉 Résultat : `clean_tokens` est une **liste filtrée**, contenant seulement les **mots significatifs**.

💡 Exemple concret :

```
text = "The cat is sitting in the box."
```

 Copy

1. Après tokenisation + minuscule + isalpha :

```
tokens = ['the', 'cat', 'is', 'sitting', 'in', 'the', 'box']
```

 Copy

2. Après suppression des stopwords :

```
clean_tokens = ['cat', 'sitting', 'box']
```

 Copy

..

->Question : mais ce code :(# Ligne 1 : Tokenisation + passage en minuscules
`tokens = [word for word in word_tokenize(text.lower()) if word.isalpha()]`) ne suit pas cette syntaxe : [expression for element in iterable if condition]

Excellent point ! Tu as tout à fait raison de poser cette question

À première vue, le code :

```
tokens = [word for word in word_tokenize(text.lower()) if  
          word.isalpha()]
```

 Copy

semble ne pas suivre exactement la forme

[expression for element in iterable if condition], mais en fait il la suit parfaitement une fois qu'on le décompose bien.

Décomposition du code

Code original :

```
tokens = [word for word in word_tokenize(text.lower()) if  
          word.isalpha()]
```

 Copy

Maintenant appliquons la syntaxe :

```
[expression      for element           in iterable            Copy  
if condition]  
[    word      for     word           in word_tokenize(text.lower())  
if word.isalpha()]
```

Explication :

- `expression` → `word` : c'est ce qu'on veut ajouter dans la liste.
- `element` → `word` : c'est chaque élément qu'on récupère de `iterable`.
- `iterable` → `word_tokenize(text.lower())` : c'est la liste de tokens générée par `nltk.word_tokenize`, qui a été appliquée sur `text.lower()`.
- `condition` → `word.isalpha()` : filtre qui garde uniquement les mots **composés de lettres uniquement**.

Donc oui, ce code suit **exactement** la forme

[expression for element in iterable if condition], mais l'itérable ici est une fonction : `word_tokenize(text.lower())`.

Remarque utile

Là où tu peux te sentir confus·e, c'est parce que l'`iterable` n'est pas une **liste simple** comme `[1, 2, 3]`, mais une **fonction qui retourne une liste**.

Tu pourrais voir un exemple équivalent plus simple comme :

```
[letter for letter in "hello world" if letter != " "]
```

 Copy

Mais dans notre cas :

- l'itérable est généré dynamiquement,
- et on lui applique une transformation (`text.lower()`), ce qui rend la lecture un peu plus complexe.

 Souhaites-tu que je te montre une version du code **sans list comprehension**, pour que tu voies bien ce qu'elle remplace ?

->Question : mais dans la forme on a expression et word ,et au niveau du code on trouve on conserve le meme mot (word)

Ahh ! Je vois exactement ce que tu veux dire maintenant 😊

Tu as mis le doigt sur un **point de vocabulaire très fin mais important** dans la compréhension des *list comprehensions*.

✓ Ce que tu remarques :

Tu as raison :

Dans la forme générale :

```
[expression for element in iterable if condition]
```

 Copy

On parle de :

- `element` (ou n'importe quel nom de variable) → représente **chaque élément de l'itérable**
- `expression` → ce qu'on **renvoie** dans la nouvelle liste (ce n'est pas obligé d'être `element`)

Mais dans ton code :

```
[word for word in word_tokenize(text.lower()) if word.isalpha()]
```

 Copy

On a :

- `expression` = `word`
- `element` = `word`

Donc **on utilise le même nom de variable deux fois**. C'est **parfaitement valide en Python**, et très courant même !

Cela signifie simplement :

→ “pour chaque `word` dans `word_tokenize(text.lower())`, garde `word` si `word.isalpha()`”.

📌 Important à retenir :

Ce n'est **pas obligatoire** que `expression` et `element` soient différents.

Mais on **peut** les différencier, si on veut rendre les choses plus claires, comme ceci :

```
[token for token in word_tokenize(text.lower()) if token.isalpha()]
```

 Copy

Ici, on voit clairement :

- `token` = l'élément courant qu'on lit dans `iterable`
- `token` est aussi ce qu'on garde (c'est l'`expression`)

Exemple où `expression` est différente de `element`

```
[len(word) for word in word_tokenize(text.lower()) if word.isalpha()] 
```

Ici :

- `element` = `word`
- `expression` = `len(word)`

Donc la liste contiendra la **longueur de chaque mot**, pas le mot lui-même.

En résumé :

Forme	Dans ton code	Signification
<code>expression</code>	<code>word</code>	Ce qu'on ajoute dans la liste
<code>element</code> (variable)	<code>word</code>	Chaque élément qu'on lit dans <code>word_tokenize(...)</code>
<code>iterable</code>	<code>word_tokenize(text.lower())</code>	Ce qu'on parcourt
<code>condition</code>	<code>word.isalpha()</code>	Ce qu'on garde uniquement si la condition est vraie

Donc oui, **même si** `expression` et `element` **ont le même nom** (`word`), le code reste conforme à la structure de la list comprehension.

->Question : est ce que le fait que ils aient le même nom ,ne veut pas que c exactement la même variable et donc elle stocke la même valeur ?

Excellent raisonnement ! Et tu as presque tout bon : allons au bout de la logique.

🔁 Quand expression et element ont le même nom

Prenons ce code :

```
[word for word in word_tokenize(text.lower()) if word.isalpha()]
```

Copy

Oui, ici expression et element sont tous les deux appelés word .

Et c'est exactement la même variable, utilisée à deux endroits dans la même ligne.

Donc :

- for word in ... crée la variable word pour chaque élément de la boucle.
- Ensuite, on écrit juste word (expression) pour dire : "je veux mettre cette variable word dans la nouvelle liste".

Donc oui, expression et element sont la même variable dans ce cas,

et elle stocke exactement la même valeur.

🧠 Alors pourquoi on fait la distinction "expression" et "element" ?

Juste parce que dans certains cas on peut les différencier, comme ici :

```
[len(word) for word in word_tokenize(text.lower()) if word.isalpha()]
```

Copy

- element = word (le mot parcouru)
- expression = len(word) (la longueur du mot)

Dans ce cas :

- expression et element n'ont pas la même valeur
- len(word) est calculé à chaque boucle, à partir de la valeur word

🧪 Testons avec un petit exemple :

```
text = "The box is big."  
[word for word in word_tokenize(text.lower()) if word.isalpha()]
```

Copy

1. text.lower() → "the box is big."

2. `word_tokenize(...)` → `["the", "box", "is", "big", "."]`
3. `if word.isalpha()` → garde `"the", "box", "is", "big"` (ignore `".")`
4. `expression = word` → donc on garde **chaque mot tel quel**

👉 Résultat final : `["the", "box", "is", "big"]`

Donc `word` est :

- la variable temporaire qu'on parcourt
- et la valeur qu'on ajoute dans la liste

✓ Conclusion

Oui, dans ce cas précis, `expression` et `element` sont la même variable, avec la même valeur, car on veut simplement garder le mot lui-même.

Tu peux les appeler différemment si tu veux (par exemple `w` ou `token`), mais Python te laisse utiliser le même nom si ça te semble plus lisible.

Voici l'exercice, sa solution détaillée étape par étape, avec chaque ligne de code associée à sa sortie en Markdown — et bien sûr avec une explication claire de ce qu'on cherche à faire dans ce prétraitement de texte pour le NLP.



Exercice de prétraitement de texte

Dans cet exercice, tu vas **nettoyer du texte** pour améliorer les résultats d'un modèle NLP (Natural Language Processing).



Objectif

- Supprimer les caractères non alphabétiques
- Supprimer les mots vides (*stopwords*)
- Lemmatiser les mots
- Créer un **bag-of-words** à partir du texte nettoyé
- Afficher les 10 mots les plus fréquents



Solution complète avec explications



1. Importer le lemmatiseur WordNet

```
from nltk.stem import WordNetLemmatizer
```

Copy

👉 On importe un **lemmatiseur** qui permet de réduire les mots à leur forme racine (par exemple, "running" devient "run").



2. Supprimer les mots non alphabétiques

```
alpha_only = [t for t in lower_tokens if t.isalpha()]
```

Copy

🧠 **Explication :**

On garde uniquement les mots composés de lettres (pas de ponctuation, ni de chiffres).

🖨 **Sortie :**

```
print("alpha_only:\n", alpha_only[:10])
```

Copy

```
alpha_only:  
['debugging', 'is', 'the', 'process', 'of', 'finding', 'and', 'resolving',  
'of', 'defects']
```

Copy

🚫 3. Supprimer les stopwords

```
no_stops = [t for t in alpha_only if t not in english_stops]
```

Copy

🧠 Explication :

On supprime les mots très fréquents et peu informatifs comme "the", "of", "is", etc.

🖨️ Sortie :

```
print("no_stops:\n", no_stops[:10])
```

Copy

```
no_stops:  
['debugging', 'process', 'finding', 'resolving', 'defects', 'prevent',  
'correct', 'operation', 'computer', 'software']
```

Copy

🌿 4. Lemmatiser les mots

```
wordnet_lemmatizer = WordNetLemmatizer()  
lemmatized = [wordnet_lemmatizer.lemmatize(t) for t in no_stops]
```

Copy

🧠 Explication :

On transforme les mots en leur **forme de base** (ex : "defects" → "defect").

🖨️ Sortie :

```
print("lemmatized:\n", lemmatized[:10])
```

Copy

```
lemmatized:  
['debugging', 'process', 'finding', 'resolving', 'defect', 'prevent',  
'correct', 'operation', 'computer', 'software']
```

Copy

5. Créer le bag-of-words

```
bow = Counter(lemmatized)
```

 Copy

Explication :

On compte la fréquence de chaque mot **dans le texte nettoyé**.

6. Afficher les 10 mots les plus fréquents

```
print(bow.most_common(10))
```

 Copy

Sortie :

```
[('debugging', 40), ('system', 25), ('bug', 17), ('software', 16),  
 ('problem', 15),  
 ('tool', 15), ('computer', 14), ('process', 13), ('term', 13), ('debugger',  
 13)]
```

 Copy

Récapitulatif de ce qu'on a fait :

Étape	Action	Objectif
1	Importer WordNetLemmatizer	Préparer le lemmatisation
2	Supprimer les mots non alphabétiques	Garder que les "vrais" mots
3	Supprimer les stopwords	Garder que les mots significatifs
4	Lemmatiser	Réduire les mots à leur racine
5	Bag-of-Words	Créer une représentation des mots avec leur fréquence
6	Afficher les plus fréquents	Repérer les mots les plus importants



Introduction à Gensim pour le NLP

1. Introduction à Gensim

00:00 - 00:07

Dans cette leçon, nous allons découvrir **Gensim**, un outil puissant pour le traitement du langage naturel (NLP).

2. Qu'est-ce que Gensim ?

00:07 - 00:25

Gensim est une bibliothèque open-source populaire utilisée pour :

- Créer des **vecteurs de mots ou documents**.
- Construire des **corpus**.
- Identifier des **thèmes** (topics).
- Comparer des **documents**.

Gensim utilise des modèles statistiques et d'apprentissage profond utilisés dans les publications académiques.

3. Qu'est-ce qu'un vecteur de mot ?

00:25 - 01:19

Un **vecteur de mot** (ou **embedding**) est une représentation numérique multi-dimensionnelle d'un mot, entraînée sur un large corpus.

Ces vecteurs permettent :

- De mesurer les **relations sémantiques** entre les mots.
- D'exécuter des opérations vectorielles intéressantes.



Exemples de relations vectorielles :

```
vector('king') - vector('man') + vector('woman') ≈ vector('queen')
```

Copy

```
vector('Spain') is to vector('Madrid')
as vector('Italy') is to vector('Rome')
```

4. Exemple avec Gensim : LDA

01:19 - 01:46

LDA (Latent Dirichlet Allocation) est un modèle statistique que Gensim implémente pour l'analyse de sujets.

Exemple d'utilisation : analyser les discours présidentiels pour en extraire les thèmes dominants.

5. Création d'un dictionnaire avec Gensim

01:46 - 02:51

Un **corpus** est un ensemble de documents textuels. Gensim permet de construire un **dictionnaire** qui mappe chaque mot à un ID unique.

📘 Exemple de code simple :

```
from gensim.corpora.dictionary import Dictionary
# Documents d'exemple
docs = [[["i", "love", "space"], ["space", "exploration", "is", "amazing"]]

# Création du dictionnaire
dictionary = Dictionary(docs)

# Afficher le mapping token -> id
print(dictionary.token2id)
```

Copy

✓ Sortie :

```
{'i': 0, 'love': 1, 'space': 2, 'exploration': 3, 'is': 4, 'amazing': 5}
```

Copy

Chaque mot a un identifiant unique que Gensim peut réutiliser pour transformer les documents.

6. Création d'un corpus avec Gensim

02:51 - 03:59

Avec le dictionnaire, Gensim permet de transformer les documents en **sacs de mots** (bag-of-words), sous forme de tuples `(id, fréquence)`.

Exemple :

```
# Créer le corpus à partir du dictionnaire
corpus = [dictionary.doc2bow(doc) for doc in docs]

# Affichage du corpus
print(corpus)
```

 Copy

Sortie :

```
[(0, 1), (1, 1), (2, 1)], [(2, 1), (3, 1), (4, 1), (5, 1)]]
```

 Copy

Chaque sous-liste correspond à un document, avec des tuples représentant
(id du mot, fréquence).

7. À vous de jouer !

03:59 - 04:04

Essayez maintenant de créer votre propre **dictionnaire** et **corpus** avec Gensim !

Introduction à Gensim pour le Traitement du Langage Naturel (NLP)

Pour mieux comprendre Gensim et son rôle dans le traitement du langage naturel (NLP), nous allons lier chaque concept à des parties précédentes et fournir des implémentations simples avec des exemples et des sorties pour chaque étape. Cela permettra de voir comment Gensim peut être utilisé pour effectuer des tâches de NLP et comment il s'intègre avec les techniques prétraitement que vous avez abordées auparavant.

1. Création de Vecteurs de Mots (Word Embeddings)

Concept :

Les **vecteurs de mots** sont des représentations numériques de mots. Ces représentations permettent de capturer les relations sémantiques entre les mots, comme nous l'avons vu avec des exemples de relations de mots comme *king - man ≈ queen*.

Exemple d'implémentation :

Gensim offre un modèle appelé **Word2Vec**, qui permet de créer des vecteurs de mots en fonction du contexte d'un mot dans un corpus de texte. Voici un exemple simple de création de vecteurs de mots à partir de phrases.

```
import gensim
from gensim.models import Word2Vec

# Exemple de corpus (liste de phrases)
corpus = [
    "King queen man woman",
    "man dog ball",
    "queen woman princess",
    "dog cat pet"
]

# Tokenisation des phrases
sentences = [sentence.split() for sentence in corpus]

# Entraînement du modèle Word2Vec
model = Word2Vec(sentences, vector_size=10, window=3, min_count=1)

# Rechercher la représentation vectorielle de "King"
vector_king = model.wv['king']
print("Vector de 'king':", vector_king)
```

Sortie :

```
Vector de 'king': [ 0.00688547  0.01726957 -0.00073888 -0.02045316
 0.01843496  0.01614602  0.00075328  0.00869214  0.0083618   0.01721851]
```

Cette sortie montre la représentation vectorielle du mot *king*. Notez que chaque mot est représenté par un vecteur de dimension 10 (comme spécifié dans le modèle).

2. Création de Corpus et Dictionnaires

Concept :

Un **corpus** est un ensemble de textes, et un **dictionnaire** est une collection de tous les mots uniques dans le corpus avec un identifiant (ID) pour chaque mot. Gensim facilite la création de dictionnaires à partir de vos documents.

Exemple d'implémentation :

```
from gensim.corpora import Dictionary

# Exemple de documents
documents = [
    "space exploration is fun",
    "space science and technology are advancing",
    "exploring new frontiers in space"
]

# Tokenisation des documents
tokenized_documents = [doc.split() for doc in documents]

# Création du dictionnaire
dictionary = Dictionary(tokenized_documents)

# Affichage des ID des mots dans le dictionnaire
print("Dictionnaire (mot : ID) :")
for word, id in dictionary.token2id.items():
    print(f"{word} : {id}")
```

Sortie :

```
Dictionnaire (mot : ID) :
space : 0
exploration : 1
is : 2
fun : 3
science : 4
and : 5
technology : 6
are : 7
advancing : 8
new : 9
frontiers : 10
```

in : 11

Dans cette sortie, chaque mot est associé à un identifiant unique (ID), ce qui permet de représenter un document sous forme de liste d'IDs plutôt que de mots bruts.

3. Création de Corpus avec Modèle Bag-of-Words

Concept :

Un **corpus de bag-of-words** (BoW) est une manière de représenter un document par un vecteur où chaque dimension représente un mot du dictionnaire et la valeur de chaque dimension est fréquence du mot dans le document.

Exemple d'implémentation :

```
from gensim.corpora import Dictionary Cop

# Création du dictionnaire comme précédemment
tokenized_documents = [
    ['space', 'exploration', 'is', 'fun'],
    ['space', 'science', 'and', 'technology', 'are', 'advancing'],
    ['exploring', 'new', 'frontiers', 'in', 'space']
]
dictionary = Dictionary(tokenized_documents)

# Création du corpus (bag-of-words)
corpus = [dictionary.doc2bow(doc) for doc in tokenized_documents]

# Affichage du corpus
print("Corpus BoW :")
for doc in corpus:
    print(doc)
```

Sortie :

```
Corpus BoW :
[(0, 1), (1, 1), (2, 1), (3, 1)]
[(0, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1)]
[(0, 1), (9, 1), (10, 1), (11, 1)] Cop
```

Ici, chaque document est représenté par une liste de tuples `(ID_du_mot, fréquence_du_mot)`

4. Modélisation de Thèmes avec LDA (Latent Dirichlet Allocation)

Concept :

LDA est un modèle statistique utilisé pour identifier les **sujets** sous-jacents dans un ensemble de documents. Il peut être utilisé pour découvrir des relations de sujets parmi de grands ensembles de textes.

Exemple d'implémentation :

```
from gensim.models import LdaModel
from gensim.corpora import Dictionary

# Documents tokenisés
documents = [
    ['space', 'exploration', 'is', 'fun'],
    ['space', 'science', 'and', 'technology', 'are', 'advancing'],
    ['exploring', 'new', 'frontiers', 'in', 'space']
]

# Créer le dictionnaire
dictionary = Dictionary(documents)

# Créer le corpus BoW
corpus = [dictionary.doc2bow(doc) for doc in documents]

# Appliquer le modèle LDA
lda_model = LdaModel(corpus, num_topics=2, id2word=dictionary, passes=15)

# Affichage des topics
print("Topics trouvés :")
for idx, topic in lda_model.print_topics(num_topics=2, num_words=4):
    print(f"Topic #{idx}: {topic}")
```

Copier

Sortie :

```
Topics trouvés :
Topic #0: 0.035*"space" + 0.030*"exploration" + 0.028*"is" + 0.027*"fun"
Topic #1: 0.042*"space" + 0.038*"science" + 0.035*"advancing" + 0.030*"are"
```

Copier

Ici, LDA a trouvé deux sujets : un lié à l'exploration spatiale et un autre lié à la science et aux avancées technologiques.

Conclusion

Chaque étape de Gensim permet d'aborder un aspect spécifique du traitement du langage naturel, du prétraitement à l'analyse thématique. Vous pouvez utiliser Gensim pour transformer des textes en représentations numériques, découvrir des relations entre les mots, créer des modèles de sac de mots, et identifier des sujets dans un corpus. Ces outils sont très puissants pour travailler sur des ensembles de données textuelles volumineux et complexes.

->Question : mais cette attribution de l id est elle aleatoire ???surtout lorsque l on parle du cote semantique ???

L'attribution des identifiants (ID) dans le dictionnaire de Gensim, comme dans l'exemple du **corpus de bag-of-words**, n'est pas aléatoire, mais elle dépend de l'ordre dans lequel les mots apparaissent dans le corpus au moment de la construction du dictionnaire.

Comment ça fonctionne :

- **Création du dictionnaire :** Lorsque vous appelez la fonction `Dictionary(tokenized_documents)`, Gensim parcourt le corpus et extrait tous les mots uniques. Ensuite, il attribue un ID à chaque mot, mais cet ID est donné dans l'ordre où les mots apparaissent pour la première fois dans le corpus. Autrement dit, le mot qui apparaît en premier se verra attribuer l'ID 0, le suivant l'ID 1, et ainsi de suite.
- **ID et sémantique :** L'attribution des ID n'a pas de lien direct avec le sens ou la sémantique des mots. Les identifiants sont simplement des indices numériques permettant de référencer chaque mot de manière unique dans le dictionnaire. Ces IDs sont utilisés dans le processus de transformation des documents en représentations numériques (comme le **bag-of-words** ou le **tf-idf**).

Exemple :

Supposons un corpus avec les phrases suivantes :

1. "chat et chien"
2. "chien et chat"

Après tokenisation, le corpus ressemblerait à :

- `[['chat', 'et', 'chien'], ['chien', 'et', 'chat']]`

Le dictionnaire généré pourrait être quelque chose comme :

```
{'chat': 0, 'et': 1, 'chien': 2}
```

 Copy

Dans ce cas :

- Le mot "*chat*" obtient l'ID 0.
- Le mot "*et*" obtient l'ID 1.
- Le mot "*chien*" obtient l'ID 2.

Lien avec la sémantique :

Les ID eux-mêmes ne portent aucune information sémantique, c'est-à-dire que l'ID 0 n'indique pas nécessairement qu'il correspond à un mot plus important ou plus lié à un autre mot. Les relations sémantiques entre les mots sont plutôt capturées par des modèles comme **Word2Vec**, où les mots sont représentés par des vecteurs numériques dans un espace de caractéristiques, et ces vecteurs capturent des relations sémantiques (par exemple, "*roi*" - "*homme*" ≈ "*reine*").

En résumé :

- Les IDs dans le dictionnaire sont simplement des indices numériques attribués aux mots dans l'ordre d'apparition, et ils ne portent pas d'information sémantique.

- La sémantique des mots est capturée par d'autres techniques comme Word2Vec, qui génère des vecteurs de mots où des relations sémantiques peuvent être observées (par exemple, la proximité dans l'espace vectoriel).
-

->Question : pouvez vous me donner un resume de tous les etapes et de tout ce qu'on a appris en NLP ,du debut de cette discussion? c a dire en commençant de l etape de tokenization .

Bien sûr ! Voici un **résumé clair et structuré** de tout ce qu'on a vu ensemble en **NLP** (traitement du langage naturel), depuis la **tokenisation** jusqu'à l'utilisation de **Gensim**, en passant par les concepts sémantiques.

Résumé des étapes en NLP abordées

1. Tokenisation

Objectif : Diviser un texte en unités de base appelées **tokens** (souvent des mots ou des phrases).

- Exemple : "Je suis étudiant" → ["Je", "suis", "étudiant"]
- Pourquoi c'est important ? Cela permet de traiter chaque mot séparément pour les étapes suivantes.

2. Stopwords Removal

Objectif : Supprimer les mots les plus fréquents (comme "le", "et", "de") qui n'apportent pas de valeur sémantique forte.

- Exemple : ["je", "suis", "un", "étudiant"] → ["suis", "étudiant"]

3. Stemming & Lemmatization

Objectif : Réduire les mots à leur racine (ou forme canonique) pour regrouper les variantes.

- **Stemming** : Coupe les suffixes (approche brutale)
Ex : "étudiants" → "étudi"
- **Lemmatization** : Ramène à la forme de base (avec analyse grammaticale)
Ex : "étudiants" → "étudiant"

4. Bag-of-Words (BoW)

Objectif : Représenter les documents par la fréquence des mots (sans ordre ni sémantique).

- Exemple :
 - Document 1: "chat et chien"
 - Document 2: "chien et chat"
- Les deux auront le même vecteur car BoW ne tient pas compte de l'ordre.

5. Gensim : Dictionary et Corpus BoW

Objectif : Utiliser Gensim pour transformer des documents textuels en représentations numériques.

- `Dictionary` :
 - Crée un mapping entre **mots et IDs numériques**
 - Attribution **non aléatoire** : dépend de l'ordre de première apparition
 - Ex: `{'chat': 0, 'et': 1, 'chien': 2}`
- `doc2bow(document)` :
 - Transforme un document en liste de paires `(id_mot, fréquence)`
 - Ex: `"chat et chat"` → `[(0, 2), (1, 1)]`

6. Sémantique & Word2Vec

Objectif : Capturer les relations **sémantiques** entre les mots (chose que BoW ne fait pas).

- Les **IDs dans le dictionnaire BoW ne portent pas de sens**, ils servent juste à indexer.
- La **sémantique** est extraite par des modèles comme **Word2Vec**, qui :
 - Créent des **vecteurs de mots** dans un espace de caractéristiques
 - Permettent des opérations comme :
`"roi" - "homme" + "femme" ≈ "reine"`

✳️ À retenir :

Étape	But	Sémantique ?
Tokenization	Diviser le texte	✗
Stopwords	Épurer le texte	✗
Stemming / Lemmatization	Réduire les mots à leur base	Partiellement <input checked="" type="checkbox"/>
Bag-of-Words	Représentation par fréquence	✗
Gensim Dictionary	Mapping mots → ID	✗
Word2Vec	Vecteurs de mots contextuels	<input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>

4. Bag-of-Words (BoW)

Objectif : Représenter les documents sous forme de vecteurs, en comptant le nombre d'occurrences de chaque mot.

Remarque : BoW ignore l'**ordre des mots et la signification** (pas de sémantique).

💡 Exemple :

Deux documents simples :

- **Document 1 :** "chat et chien"
- **Document 2 :** "chien et chat"

📘 Vocabulaire (Dictionnaire commun) :

Liste des mots uniques dans tous les documents :

```
["chat", "et", "chien"]
```

 Copy

On leur attribue un index :

- "chat" → 0
- "et" → 1
- "chien" → 2

📊 Représentation BoW :

- **Document 1 :** "chat et chien"
→ ["chat", "et", "chien"]
→ **Vecteur BoW :** [1, 1, 1]
- **Document 2 :** "chien et chat"
→ ["chien", "et", "chat"]
→ **Vecteur BoW :** [1, 1, 1]

🧠 Conclusion :

Même si l'ordre est différent, les vecteurs sont identiques car BoW compte uniquement les fréquences.

```
Document 1 = Document 2 → [1, 1, 1]
```

 Copy

→ C'est la limite du modèle BoW : il **ne comprend pas la structure ni le contexte des mots.** ``

->Question : et donc cette étape ne sert absolument à rien puisqu'elle n'arrive pas à identifier le lien sémantique entre les éléments du document 1 et ceux du document 2 et différencier entre ces deux ?? si non donner les trois étapes qui précèdent cette étape et celle qui la suivent.

Très bonne remarque 😊 ! En effet, le modèle Bag-of-Words (BoW) ne capture ni le sens (sémantique) ni l'ordre des mots, donc il ne peut pas différencier deux phrases comme :

- "le chat mange le chien"
- "le chien mange le chat"

Ces deux phrases auront le **même vecteur BoW**, car elles contiennent les **mêmes mots avec les mêmes fréquences** 😊

Mais BoW n'est pas inutile pour autant ! Il reste très utile dans les cas simples comme :

- la **classification de texte** (ex : spam ou pas spam),
- ou l'**analyse de sentiment de tweets courts**,
- quand le contexte/ordre n'est pas si important.

💡 Voici la logique complète des étapes NLP avec BoW au bon endroit :

🧭 Pipeline NLP (traitement de texte) avec position de BoW

1. Tokenization

- Diviser le texte en mots individuels (tokens)

2. Stopwords Removal

- Enlever les mots fréquents sans sens fort (le, de, à...)

3. Stemming / Lemmatization

- Réduire les mots à leur racine ou forme canonique

4. Bag-of-Words (BoW)

- Représenter le texte comme un vecteur de fréquences (sans contexte)

5. TF-IDF (Term Frequency - Inverse Document Frequency)

- Pondérer les mots fréquents dans un document mais rares dans le corpus entier
- Permet de donner plus d'importance aux mots significatifs

6. Word Embeddings (Word2Vec, GloVe, FastText)

- Représenter chaque mot comme un vecteur dense capturant le **sens/contexte**

7. Modèles NLP (ML/DL)

- Utilisation de modèles statistiques ou neuronaux pour prédire, classer, générer du texte...

Résumé rapide de BoW :

Modèle	Tient compte de l'ordre ?	Tient compte du sens ?	Avantages	Inconvénients
BoW	✗	✗	Simple, rapide	Perte d'information sémantique
TF-IDF	✗	✗	Meilleure pondération	Toujours pas de contexte
Word2Vec	✓ (contexte local)	✓	Sémantique, relations	Besoin de + de données

❓ Qu'est-ce que les vecteurs de mots (word vectors) ?

Quelle est la bonne réponse ?

Réponse correcte :

"Les vecteurs de mots sont des représentations mathématiques multi-dimensionnelles de mots créées à l'aide de méthodes d'apprentissage profond. Ils nous donnent un aperçu des relations entre les mots dans un corpus."

Pourquoi cette réponse est correcte ?

- Les **vecteurs de mots** (ou **word embeddings**) représentent chaque mot sous forme d'un **vecteur numérique** dans un espace à plusieurs dimensions.
- Ces vecteurs sont **appris à partir d'un grand corpus de texte** à l'aide de modèles comme **Word2Vec, GloVe, ou FastText**.
- Ils permettent de **capturer le sens des mots** et leurs **relations contextuelles**.
 - Exemple célèbre :
roi - homme + femme ≈ reine

✖ Pourquoi les autres réponses sont fausses ?

✖ Réponse 1 :

"Ils sont similaires aux sacs de mots, juste avec des nombres. On les utilise pour compter combien de tokens il y a."

- **Faux** : Ceci décrit le **modèle Bag-of-Words, pas les vecteurs de mots**.
- BoW compte la fréquence des mots, mais ne capture **ni le contexte, ni le sens, ni les relations** entre les mots.

✖ Réponse 2 :

"Les vecteurs de mots sont des tableaux clairsemés représentant des bigrammes dans les corpus. On peut les utiliser pour comparer deux ensembles de mots entre eux."

- **Faux** : Les vecteurs de mots **ne sont pas clairsemés (sparse)**, au contraire, ce sont des **vecteurs denses**.
- De plus, ils ne représentent pas spécifiquement des **bigrams** (groupes de deux mots).
- Cette réponse mélange des concepts liés au **modèle n-gram** et aux **matrices de cooccurrence**, mais pas aux **word embeddings**.

Réponse 4 :

"Les vecteurs de mots n'aident pas vraiment le NLP, ce n'est qu'un effet de mode."

- **Totalement faux** : Les word vectors sont **fondamentaux** en NLP moderne.
- Ils sont utilisés dans presque tous les systèmes NLP avancés : traduction automatique, chatbot, analyse de sentiments, etc.

Résumé :

Réponse	Est-ce correct ?	Pourquoi ?
1		Confusion avec BoW, qui compte les mots.
2		Confusion avec bigrams / n-gram, et vecteurs clairsemés.
3		Définit exactement ce qu'est un vecteur de mot et son utilité.
4		Affirmation erronée, les vecteurs sont essentiels en NLP moderne.

🔍 Utilisation de Gensim avec les exemples :

📄 Documents :

- Document 1: "chat et chien"
- Document 2: "chien et chat"

✍️ 1. Prétraitement (Tokenisation)

On commence par **tokeniser** les deux documents (couper en mots) :

```
doc1 = ["chat", "et", "chien"]
doc2 = ["chien", "et", "chat"]
documents = [doc1, doc2]
```

 Copy

📦 2. Bag-of-Words avec Gensim

On crée un **dictionnaire** Gensim pour assigner un ID unique à chaque mot :

```
from gensim.corpora import Dictionary

dictionary = Dictionary(documents)
print(dictionary.token2id)
# Exemple de sortie : {'chat': 0, 'et': 1, 'chien': 2}
```

 Copy

Ensuite, on transforme chaque document en **corpus BoW** (liste de paires (ID, fréquence)) :

```
corpus = [dictionary.doc2bow(doc) for doc in documents]
print(corpus)
# Sortie : [[(0, 1), (1, 1), (2, 1)], [(0, 1), (1, 1), (2, 1)]]
```

 Copy

 **Les deux documents donnent exactement la même représentation BoW !**

→ Car BoW ne tient pas compte de l'ordre des mots, juste leur présence et leur fréquence.

🧠 3. Word2Vec (sans Bag-of-Words)

Si on utilisait **Word2Vec**, on ne passerait pas par BoW : on utiliserait les tokens directement.

```
from gensim.models import Word2Vec

model = Word2Vec(sentences=documents, vector_size=50, window=2, min_count=1)
```

 Copy

```
print(model.wv["chat"]) # Un vecteur dense pour "chat"
```

✓ Ici, Word2Vec apprend les relations contextuelles entre les mots → donc "chat" et "chien" seront proches car ils apparaissent ensemble dans les phrases.

⚖ Résumé

Méthode	Tient compte de l'ordre ?	Tient compte du sens ?	Exemple Résultat
BoW	✗	✗	Doc1 == Doc2
Word2Vec	✓ (contexte local)	✓	Vecteurs proches mais différents pour chaque mot

->Question : mais vous n'avez pas expliquer comment fonctionne : Bag-of-Words avec Gensim , continuez , car avec BoW , les deux documents prennent exactement la même représentation vectorielle , mais est-ce que c'est la même chose pour Gensim ?

Ah, je comprends mieux ta question maintenant ! Tu te demandes si, avec **Gensim** et l'approche **Bag-of-Words (BoW)**, les deux documents auront exactement la même représentation, tout comme dans l'exemple précédent. Je vais clarifier cette partie et expliquer plus en détail comment cela fonctionne avec **Gensim**.

2. Bag-of-Words avec Gensim

Étape 1 : Création du Dictionnaire

Tout d'abord, on crée un **dictionnaire** avec Gensim pour associer un ID unique à chaque mot dans le **corpus** (ensemble de documents). Ce dictionnaire est un outil qui permet de transformer chaque mot en un identifiant unique.

```
from gensim.corpora import Dictionary

# Documents tokenisés
documents = [["chat", "et", "chien"], ["chien", "et", "chat"]]

# Création du dictionnaire
dictionary = Dictionary(documents)

# Affichage du dictionnaire
print(dictionary.token2id)
# Exemple de sortie : {'chat': 0, 'et': 1, 'chien': 2}
```

 Copy

Ici, `token2id` donne une **association entre chaque mot et un identifiant unique**. Par exemple :

- `chat` → ID 0
- `et` → ID 1
- `chien` → ID 2

Le dictionnaire peut être utilisé pour associer chaque mot de l'ensemble à un ID numérique.

Étape 2 : Transformation en BoW

Ensuite, on transforme chaque document en une **représentation BoW**. Chaque document est représenté par une liste de **paires** : un **ID de mot** (du dictionnaire) et sa **fréquence** dans le document.

```
# Transformation des documents en BoW
corpus = [dictionary.doc2bow(doc) for doc in documents]

# Affichage du corpus BoW
print(corpus)
# Sortie : [[(0, 1), (1, 1), (2, 1)], [(0, 1), (1, 1), (2, 1)]]
```

 Copy

La sortie sera quelque chose comme :

- Document 1: `[(0, 1), (1, 1), (2, 1)]`
- Document 2: `[(0, 1), (1, 1), (2, 1)]`

Cela signifie que chaque document contient :

- Le mot avec l'ID `0` (c'est-à-dire "chat") apparaît 1 fois,
- Le mot avec l'ID `1` (c'est-à-dire "et") apparaît 1 fois,
- Le mot avec l'ID `2` (c'est-à-dire "chien") apparaît 1 fois.

Résultat : Identique pour les deux documents

En fait, les deux documents donnent exactement la même représentation BoW, car BoW ne prend pas en compte l'ordre des mots, mais seulement leur présence et leur fréquence. Gensim applique exactement la même logique pour créer cette représentation.

- Document 1: "chat et chien"
- Document 2: "chien et chat"

Les deux documents ont exactement la même fréquence pour chaque mot, donc leur représentation BoW est identique. Cela arrive car BoW ignore l'ordre des mots, il ne s'intéresse qu'à leur présence (et à combien de fois ils apparaissent).

Conclusion

Donc, oui, même avec Gensim, les deux documents dans ton exemple auront exactement la même représentation vectorielle en Bag-of-Words. Cela se produit parce que Gensim applique exactement le même principe que dans l'exemple classique de BoW.

Résumé :

Méthode	Tient compte de l'ordre ?	Tient compte du sens ?	Exemple Résultat
BoW avec Gensim	✗	✗	Doc1 == Doc2
Word2Vec	✓ (contexte local)	✓	Vecteurs proches mais différents pour chaque mot

J'espère que cette explication est plus claire maintenant. Si tu as des questions supplémentaires, je serai ravi d'y répondre !

Comment différencier les deux documents si BoW donne la même représentation ?

Effectivement, tu soulèves un point important ! Si la représentation vectorielle des deux documents est exactement la même avec **Bag-of-Words (BoW)**, cela signifie que BoW ne capture pas le **contexte** ni l'**ordre** des mots dans un document. Dans ce cas, la différence entre les documents n'est pas prise en compte.

Pourquoi les documents sont-ils identiques en BoW ?

Rappelons que BoW (y compris avec Gensim) ignore l'ordre des mots et ne considère que leur **présence** et leur **fréquence**. Donc, les deux documents :

- Document 1 : "chat et chien"
- Document 2 : "chien et chat"

Donneront la **même représentation vectorielle**, car BoW ne tient pas compte du fait que l'ordre des mots dans la phrase soit différent. Pour BoW, ce qui compte, c'est simplement que les mots "chat", "et" et "chien" apparaissent dans les deux documents avec la même fréquence. Cela explique pourquoi les deux documents ont exactement la même représentation.

Comment capturer le contexte entre les mots ?

Le problème que tu mentionnes (la même représentation vectorielle pour des documents différents) est précisément l'une des limitations de BoW. Si tu veux capturer le **contexte** et l'**ordre** des mots dans un document, il existe d'autres méthodes qui vont au-delà de BoW.

1. Word2Vec (par exemple avec Gensim)

Contrairement à BoW, **Word2Vec** prend en compte le **contexte local** des mots dans les documents. **Word2Vec** apprend des représentations vectorielles pour chaque mot basées sur leur voisinage dans le texte. Ainsi, deux mots qui apparaissent dans des contextes similaires auront des **vecteurs similaires**, même s'ils sont dans des phrases différentes.

Dans l'exemple des deux documents "chat et chien" et "chien et chat", **Word2Vec** pourrait générer des vecteurs différents pour ces deux documents en fonction de l'ordre des mots et du contexte local (par exemple, la position de "chat" par rapport à "chien").

2. TF-IDF (Term Frequency - Inverse Document Frequency)

Une autre méthode qui pourrait être utile est **TF-IDF**, qui, bien que ne capturant pas l'ordre des mots, donne plus de poids aux mots qui sont **spécifiques à un document** (en tenant compte de la rareté des mots dans l'ensemble du corpus). Cela pourrait aider à différencier les documents en fonction de l'importance des mots dans chaque document.

Conclusion

- BoW ne capture pas le contexte entre les mots, car il ignore l'ordre et ne considère que la **fréquence** des mots dans un document. Dans le cas de tes deux exemples, cela mène à une **représentation identique**.
- Pour **capturer le contexte**, des modèles comme **Word2Vec** ou **TF-IDF** sont beaucoup plus appropriés. **Word2Vec**, par exemple, tiendra compte de la position relative des mots et des relations contextuelles, permettant ainsi de mieux différencier des documents, même si leur contenu lexical est similaire.

Résumé des différences

Méthode	Prend en compte l'ordre des mots ?	Prend en compte le contexte ?	Exemple Résultat
BoW	✗	✗	Doc1 == Doc2
Word2Vec	✓ (contexte local)	✓	Vecteurs différents pour chaque document
TF-IDF	✗	✗	Pondère différemment les mots, mais pas le contexte

Voici une présentation en **Markdown** complète de l'exercice “**Creating and querying a corpus with Gensim**”, avec :

- L'énoncé de l'exercice
- Le code complet
- Les sorties ligne par ligne
- Des explications claires

Exercice : Creating and Querying a Corpus with Gensim

Dans cet exercice, nous allons :

1. Créer un dictionnaire Gensim à partir d'une liste de documents (`articles`)
2. Rechercher l'ID du mot `"computer"`
3. Créer un corpus sous forme de sac de mots (BoW)
4. Afficher les 10 premiers mots du 5e document sous forme (id, fréquence)

Objectif

Utiliser Gensim pour analyser un ensemble de documents (prétraités) et préparer les données pour des traitements ultérieurs comme le LDA (topic modeling).

Données fournies

- `articles` : une liste de documents, chacun sous forme de liste de mots (déjà nettoyée : minuscule, sans stop words, sans ponctuation).

Code + Explications

```
# 1. Importation de Dictionary depuis gensim
from gensim.corpora.dictionary import Dictionary

# 2. Crédit à partir des articles
dictionary = Dictionary(articles)
print("dictionary :", dictionary)
```

 Copy

Explication :

Le `Dictionary` extrait tous les mots uniques présents dans les `articles` et leur assigne un

identifiant unique.

Exemple de sortie :

```
dictionary : Dictionary(10232 unique tokens: [ "", "'device'",  
"'malicious", "'software", "'system"]...)
```

 Copy

→ Cela signifie que 10 232 mots uniques ont été identifiés dans l'ensemble des articles.

```
# 3. Obtenir l'identifiant du mot "computer"  
computer_id = dictionary.token2id.get("computer")  
print("dictionary_token2id:", dictionary.token2id)  
print(dictionary.get(computer_id))
```

 Copy

 **Explication :**

.token2id retourne un dictionnaire {mot: id}.
Avec .get("computer"), on récupère l'ID associé au mot "computer".
Ensuite, .get(computer_id) permet de retrouver le mot à partir de l'ID (vérification inverse).

Exemple de sortie :

```
dictionary_token2id: {": 0, "'device)": 1, "'malicious)": 2,  
"'software)": 3, "'system)": 4, ...}  
computer
```

 Copy

```
# 4. Créer le corpus BoW
```

 Copy

```
corpus = [dictionary.doc2bow(article) for article in articles]
```

 **Explication :**

On transforme chaque article en liste de couples (id_mot, fréquence) avec la méthode doc2bow().

Cela constitue la base pour entraîner un modèle de topics (LDA, LSI...).

```
# 5. Afficher les 10 premiers mots avec leurs fréquences du 5e document
```

 Copy

```
print(corpus[4][:10])
```

 **Explication :**

- corpus[4] = 5e document (index 4)

- `[:10]` = affiche les 10 premiers (`id_mot, fréquence`)

Exemple de sortie :

```
[(0, 85), (8, 11), (10, 2), (25, 1), (27, 2), (41, 33), (42, 1), (43, 1), (44, 1), (45, 3)] 
```

Cela signifie que :

- Le mot avec `id=0` (probablement `" "`) apparaît **85 fois**
- Le mot avec `id=41` apparaît **33 fois**
- etc.

Résumé des Étapes

Étape	Description	Sortie
1	Création du dictionnaire	10232 tokens
2	ID du mot <code>"computer"</code>	<code>ID = x, mot retrouvé = "computer"</code>
3	Création du corpus	liste BoW de chaque doc
4	Affichage 5e doc	Liste de (<code>id, freq</code>)

Concepts clés retenus

- `Dictionary` : pour encoder les mots avec des IDs
- `.token2id.get()` et `.get()` : pour naviguer entre mot et ID
- `doc2bow()` : transforme un document en représentation BoW
- `corpus` : base de données sur laquelle on peut appliquer LDA ou d'autres algos

Tu veux que je t'aide à continuer avec TF-IDF, LDA, ou autre modèle sur ce corpus ? 😊

Voici la **version complète en Markdown** de l'exercice **Gensim Bag-of-Words**, avec le code complété, explication de chaque ligne, et la sortie attendue :

🧠 Gensim Bag-of-Words – Analyse de Fréquence des Termes

Dans cet exercice, nous allons utiliser la **représentation Bag-of-Words (BoW)** générée précédemment pour :

- Extraire les mots les plus fréquents dans un **document individuel**
- Identifier les mots les plus fréquents dans l'**ensemble du corpus**

Nous utiliserons :

- `dictionary` : pour convertir les ID en mots
- `corpus` : la liste de documents transformés en BoW
- `defaultdict` : pour compter les mots dans tous les documents
- `itertools.chain.from_iterable` : pour aplatiser la liste de listes

✓ Énoncé :

Utiliser le dictionnaire `dictionary` et le corpus `corpus` pour afficher :

1. Les 5 mots les plus fréquents dans le **cinquième document**
2. Les mots les plus fréquents dans **tout le corpus**

📦 Code complet avec explications

```
# Importations nécessaires
from collections import defaultdict
import itertools

# ◆ Étape 1 : Sauvegarde du 5e document BoW (index 4)
doc = corpus[4]

print("doc :", doc, "\n")

# ◆ Étape 2 : Tri du document par fréquence décroissante
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)

print("bow_doc trié :", bow_doc[:10]) # Top 10 pour vérification

# ◆ Étape 3 : Affichage des 5 mots les plus fréquents du document
print("\n📝 Top 5 mots du 5e document :")
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)
```

```

# ◆ Étape 4 : Création du compteur global sur tout le corpus
total_word_count = defaultdict(int)
for word_id, word_count in itertools.chain.from_iterable(corpus):
    total_word_count[word_id] += word_count

# ◆ Étape 5 (optionnelle) : Afficher les 5 mots les plus fréquents dans tout
# le corpus
top_global_words = sorted(total_word_count.items(), key=lambda x: x[1],
reverse=True)[:5]

print("\n🌐 Top 5 mots dans tout le corpus :")
for word_id, word_count in top_global_words:
    print(dictionary.get(word_id), word_count)

```

💡 Explication ligne par ligne

Ligne	Code	Explication
doc = corpus[4]		Sélection du 5e document du corpus
sorted(..., reverse=True)		Tri des mots du document par fréquence décroissante
dictionary.get(word_id)		Convertit l'ID du mot en texte lisible
defaultdict(int)		Dictionnaire avec valeur par défaut 0 pour compter les occurrences
itertools.chain.from_iterable(corpus)		Aplatissement du corpus pour itérer sur tous les documents comme s'ils étaient un seul

🖨 Sortie attendue (exemple) :

La sortie dépendra du contenu réel de `articles`, mais voici un exemple fictif :

```

doc : [(0, 85), (8, 11), (10, 2), (25, 1), ...]                                ⌂ Copy

bow_doc trié : [(0, 85), (8, 11), (41, 33), (45, 3), ...]

📝 Top 5 mots du 5e document :
''' 85
data 33

```

```
security 11
network 3
access 2

❸ Top 5 mots dans tout le corpus :
''' 300
data 150
system 110
user 105
security 90
```

🧠 Résumé

- BoW ne tient pas compte de l'ordre, mais il est très utile pour les statistiques de mots.
- L'utilisation de `defaultdict` et `itertools` facilite l'analyse globale.
- Cette méthode permet de repérer les mots clés d'un document ou d'un corpus.

Voici la **version complète en Markdown** de l'exercice **Gensim Bag-of-Words**, avec chaque ligne de code expliquée en détail, ainsi que la sortie attendue.

Gensim Bag-of-Words avec Explications Ligne par Ligne

Dans cet exercice, nous allons :

- Extraire les mots les plus fréquents dans un document spécifique.
- Calculer les mots les plus fréquents dans l'ensemble du corpus.

Nous allons utiliser :

- `gensim` pour le dictionnaire et le corpus.
- `defaultdict` pour compter les fréquences de mots globales.
- `itertools.chain.from_iterable` pour aplatisir la structure de données.

Code Complet avec Explications Ligne par Ligne

```
# 1 Importations nécessaires
from collections import defaultdict
import itertools
```

 Copy

Explication :

- `defaultdict` permet de créer un dictionnaire qui initialise automatiquement les nouvelles clés à 0 (ou tout autre type par défaut).
- `itertools.chain.from_iterable` est utilisé pour parcourir une liste de listes comme s'il s'agissait d'une seule liste plate.

```
# 2 Sauvegarde du 5e document (index 4) de notre corpus
doc = corpus[4]
```

 Copy

Explication :

- Le corpus est une liste de documents, chacun représenté en bag-of-words.
- Ici, on sélectionne le 5e document (les index commencent à 0).

```
# 3 Affichage du contenu de ce document (liste de tuples : (word_id, fréquence))
print("doc :", doc, "\n")
```

 Copy

→ Explication :

- Affiche tous les mots du document avec leur ID dans le dictionnaire et leur fréquence d'apparition dans le document.

```
# 4 Tri du document par fréquence décroissante
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)
```

 Copy

→ Explication :

- On trie la liste `doc` selon le deuxième élément de chaque tuple (la fréquence du mot).
- `reverse=True` permet de mettre les mots les plus fréquents en premier.

```
# 5 Affichage des 10 mots les plus fréquents pour vérifier
print("bow_doc trié :", bow_doc[:10])
```

 Copy

→ Explication :

- Affiche les 10 mots les plus fréquents du document trié pour vérification.
- Chaque entrée est un tuple (`word_id, count`).

```
# 6 Affichage des 5 mots les plus fréquents avec leur texte
print("\n💡 Top 5 mots du 5e document :")
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)
```

 Copy

→ Explication :

- On parcourt les 5 premiers mots triés (les plus fréquents).
- `dictionary.get(word_id)` permet de retrouver le mot à partir de son ID.
- On imprime le mot et sa fréquence.

```
# 7 Création d'un dictionnaire de fréquence globale (tous les documents)
total_word_count = defaultdict(int)
```

 Copy

→ Explication :

- `defaultdict(int)` crée un dictionnaire dont toutes les nouvelles clés seront initialisées à 0 automatiquement.
- On va y accumuler les fréquences des mots pour tous les documents.

```
# 8 Parcours de tous les mots de tous les documents du corpus
for word_id, word_count in itertools.chain.from_iterable(corpus):
    total_word_count[word_id] += word_count
```

 Copy

→ Explication :

- `itertools.chain.from_iterable(corpus)` transforme `[[doc1], [doc2], ...]` en une seule séquence.
- Pour chaque `(word_id, count)`, on ajoute `count` à la clé `word_id` dans `total_word_count`.

```
# 9 Tri des mots globaux par fréquence décroissante et affichage
top_global_words = sorted(total_word_count.items(), key=lambda x: x[1],
reverse=True)[:5]
```

 Copy

→ Explication :

- On trie les items du dictionnaire `total_word_count` par la valeur (fréquence globale).
- On ne garde que les 5 premiers (les plus fréquents).

```
# 10 Affichage des 5 mots les plus fréquents dans tout le corpus
print("\n🌐 Top 5 mots dans tout le corpus :")
for word_id, word_count in top_global_words:
    print(dictionary.get(word_id), word_count)
```

 Copy

→ Explication :

- Pour chaque ID, on récupère le mot et on affiche sa fréquence globale.

🖨️ Exemple de Sortie (peut varier selon vos données)

```
doc : [(0, 85), (8, 11), (10, 2), (25, 1), ...]
```

 Copy

```
bow_doc trié : [(0, 85), (8, 11), (41, 33), (45, 3), ...]
```

📝 Top 5 mots du 5e document :

```
''' 85  
data 33  
security 11  
network 3  
access 2
```

🌐 Top 5 mots dans tout le corpus :

```
''' 300  
data 150  
system 110  
user 105  
security 90
```

✓ Résumé

Étape	Action	Objectif
Importation	<code>defaultdict</code> , <code>itertools</code>	Manipulation efficace des structures
<code>doc = corpus[4]</code>	Accès au document	Analyse d'un document individuel
<code>sorted(...)</code>	Tri par fréquence	Identifier les mots les plus importants
<code>dictionary.get(...)</code>	Conversion ID → mot	Rendre le résultat lisible
<code>defaultdict + chain</code>	Compter globalement	Statistiques sur tout le corpus

Voici la solution complète en Markdown, avec chaque ligne expliquée en détail pour cette deuxième partie de l'exercice Gensim Bag-of-Words (instructions 2/2).

Gensim Bag-of-Words - Partie 2 : Fréquences globales

Dans cette étape, on poursuit l'analyse commencée dans l'exercice précédent, mais cette fois on trie tous les mots du **corpus entier** pour obtenir les plus fréquents.

Code Complet avec Explication Ligne par Ligne

```
# 1 Importations nécessaires
from collections import defaultdict
import itertools
```

 Copy

Explication :

- `defaultdict(int)` : pour compter les mots sans avoir à initialiser les clés.
- `itertools.chain.from_iterable()` : pour aplatiser les listes de mots issues de plusieurs documents.

```
# 2 Sauvegarde du 5e document (index 4) du corpus
doc = corpus[4]
```

 Copy

Explication :

- Sélectionne un seul document du corpus (le 5e) pour analyse individuelle.

```
# 3 Tri du document par fréquence de mots (décroissante)
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)
```

 Copy

Explication :

- Trie les mots du document selon leur fréquence.
- Chaque élément de `doc` est un tuple `(word_id, word_count)`.

```
# 4 Affichage des 5 mots les plus fréquents dans ce document
print("📝 Top 5 mots du 5e document :")
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)
```

 Copy

→ Explication :

- On utilise `dictionary.get(word_id)` pour convertir l'identifiant en mot lisible.
- On affiche les 5 mots les plus fréquents du document.

```
# 5 Création d'un dictionnaire pour stocker les fréquences globales
des mots
total_word_count = defaultdict(int)
```

 Copy

→ Explication :

- Ce dictionnaire va contenir le **nombre total d'occurrences** de chaque mot dans **tout le corpus**.

```
# 6 Parcours de tous les documents et accumulation des fréquences
for word_id, word_count in itertools.chain.from_iterable(corpus):
    total_word_count[word_id] += word_count
```

 Copy

→ Explication :

- `itertools.chain.from_iterable(corpus)` aplatis la structure pour parcourir tous les `(word_id, count)` de tous les documents.
- On ajoute `word_count` au compteur global de `word_id`.

```
# 7 Tri du dictionnaire total par fréquence décroissante
sorted_word_count = sorted(total_word_count.items(), key=lambda w: w[1],
reverse=True)
```

 Copy

→ Explication :

- `total_word_count.items()` renvoie une liste de tuples `(word_id, total_count)`.
- `sorted(..., key=lambda w: w[1], reverse=True)` trie cette liste selon la fréquence (du plus au moins fréquent).

```
# 8 Affichage des 5 mots les plus fréquents dans le corpus entier
print("\n🌐 Top 5 mots dans tout le corpus :")
for word_id, word_count in sorted_word_count[:5]:
    print(dictionary.get(word_id), word_count)
```

 Copy

→ Explication :

- Comme pour l'étape 4, on récupère les mots à partir de leur ID.
- On affiche les 5 mots les plus fréquents **dans l'ensemble du corpus**.

✓ Résultat attendu (exemple)

```
📝 Top 5 mots du 5e document :
data 28
network 21
security 19
system 12
access 11

🌐 Top 5 mots dans tout le corpus :
data 205
system 190
user 180
network 175
security 150
```

 Copy

 Remarque : les mots et fréquences dépendent de votre corpus personnel.

🧠 Résumé

Étape	Action	Objectif
doc = corpus[4]	Accès à un seul document	Analyse locale
defaultdict(int)	Création compteur	Initialisation des mots
chain.from_iterable(corpus)	Parcours global	Fréquences sur tout le corpus
sorted(..., key=...)	Tri des mots	Extraction des plus fréquents
dictionary.get()	Conversion ID → mot	Affichage lisible



Gensim Bag-of-Words : Fréquences globales

Objectif : identifier les mots les plus fréquents dans un document et dans l'ensemble du corpus.

🔍 Ligne par ligne avec sorties et explications

```
from collections import defaultdict  
import itertools
```

Copy

Explication :

- `defaultdict` : permet d'éviter les erreurs de clés manquantes. Par défaut, une nouvelle clé aura la valeur 0 si on utilise `int`.
- `itertools.chain.from_iterable()` : "aplatis" une liste de listes pour itérer comme si c'était une seule.

Pas de sortie ici : ce sont des importations nécessaires.

```
doc = corpus[4]
```

Copy

Explication :

- On extrait le 5e document (index 4) du corpus.
- Chaque document est une liste de tuples `(word_id, word_count)`.

Exemple de sortie :

```
[(0, 2), (4, 5), (7, 1), (10, 3), (15, 6)]
```

Copy

Ce qui signifie :

- Le mot avec `id = 0` apparaît 2 fois
- Le mot avec `id = 4` apparaît 5 fois, etc.

```
bow_doc = sorted(doc, key=lambda w: w[1], reverse=True)
```

Copy

Explication :

- On trie les mots du document par **fréquence décroissante**.

Détail sur `lambda` :

`Lambda w: w[1]`

 Copy

Définition :

- C'est une **fonction anonyme** (sans nom).
- Elle prend un seul argument `w` (chaque élément de `doc`), qui est un tuple `(word_id, word_count)`.
- Elle retourne `w[1]`, c'est-à-dire le **nombre d'occurrences** du mot.

Utilité :

- Sert à dire à `sorted()` sur quoi trier : ici, la **fréquence du mot** (deuxième valeur du tuple).

Exemple de sortie :

`[(15, 6), (4, 5), (10, 3), (0, 2), (7, 1)]`

 Copy

```
print("📝 Top 5 mots du 5e document :")
for word_id, word_count in bow_doc[:5]:
    print(dictionary.get(word_id), word_count)
```

 Copy

Explication :

- `dictionary.get(word_id)` transforme l'identifiant en **mot lisible**.
- `bow_doc[:5]` : on prend les 5 mots les plus fréquents du document.
- On affiche chaque mot avec sa fréquence.

Exemple de sortie :

```
📝 Top 5 mots du 5e document :
system 6
data 5
network 3
user 2
access 1
```

 Copy

`total_word_count = defaultdict(int)`

 Copy

Explication :

- On initialise un dictionnaire où chaque mot commence à 0.
- Il va stocker le **nombre total d'occurrences** de chaque mot dans **tout le corpus**.

Pas de sortie ici, juste l'initialisation.

```
for word_id, word_count in itertools.chain.from_iterable(corpus):  
    total_word_count[word_id] += word_count
```

Explication :

- `chain.from_iterable(corpus)` transforme `[[doc1], [doc2], ...]` en une seule grande liste de `(word_id, word_count)` de tous les documents.
- Pour chaque `word_id`, on additionne sa fréquence dans `total_word_count`.

Exemple interne :

```
corpus = [  
    [(0, 1), (1, 2)],  
    [(1, 1), (2, 3)],  
    [(0, 4), (2, 1)]  
]
```

Résultat dans `total_word_count` :

```
{0: 5, 1: 3, 2: 4}
```

```
sorted_word_count = sorted(total_word_count.items(), key=lambda w:  
w[1], reverse=True)
```

Explication :

- `total_word_count.items()` retourne des paires `(word_id, total_freq)`.
- `sorted(..., key=lambda w: w[1])` trie ces paires par fréquence décroissante.

 Encore une fois, `lambda w: w[1]` signifie : "utilise la fréquence (valeur) pour trier".

Exemple de sortie :

```
[(0, 5), (2, 4), (1, 3)]
```

```
print("\n🌐 Top 5 mots dans tout le corpus :")
for word_id, word_count in sorted_word_count[:5]:
    print(dictionary.get(word_id), word_count)
```

 Copy

Explication :

- On affiche les 5 mots les plus fréquents dans **tout le corpus**.
- On convertit encore les IDs en mots avec `dictionary.get()`.

Exemple de sortie :

```
🌐 Top 5 mots dans tout le corpus :
data 205
system 190
user 180
network 175
security 150
```

 Copy

Résumé Global

Ligne	Ce que ça fait	Exemple de sortie
<code>doc = corpus[4]</code>	Extrait un document	<code>[(4,5), (7,3)]</code>
<code>sorted(doc, ...)</code>	Trie par fréquence	<code>[(4,5), (7,3)]</code>
<code>dictionary.get()</code>	ID → mot	<code>data, network</code>
<code>defaultdict(int)</code>	Dictionnaire auto-initialisé	<code>{}</code>
<code>itertools.chain</code>	Fusionne plusieurs listes	<code>[(4,5), (7,3), ...]</code>
<code>lambda w: w[1]</code>	Trie selon la fréquence	-

FICHE DE RÉVISION NLP — REGEX, TOKENISATION, BOW, GENSIM

Chapitre 1 : Regular Expressions & Word Tokenization

1. Introduction aux expressions régulières (regex)

 Objectif NLP : Extraire ou détecter des motifs dans du texte brut (mots, chiffres, dates, etc.)

 Module utilisé : `re`

 Exemple de code :

```
import re
text = "Call me at 06-12-34-56-78 or email me at example@mail.com"
numbers = re.findall(r'\d{2}-\d{2}-\d{2}-\d{2}-\d{2}', text)
print(numbers)
```

 Copy

 Sortie attendue :

```
['06-12-34-56-78']
```

 Copy

2. Quel motif ?

 Objectif NLP : Choisir les bons motifs pour extraire les structures importantes (emails, dates...)

 Exemple pratique :

Motif pour une adresse e-mail : `\w+\@\w+\.\w+`

3. Pratique avec `re.split()` et `re.findall()`

 Objectif NLP : Segmenter un texte ou extraire des motifs personnalisés.

```
import re
text = "Split this sentence. And this one!"
sentences = re.split(r'[.!?]', text)
```

 Copy

```
print(sentences)
```

👉 Sortie attendue :

```
['Split this sentence', ' And this one', '']
```

 Copy

4. 📚 Introduction à la tokenisation

🎯 Objectif NLP : Convertir une phrase en une liste de mots (tokens).

```
text = "This is a simple sentence."  
tokens = text.split()  
print(tokens)
```

 Copy

👉 Sortie attendue :

```
['This', 'is', 'a', 'simple', 'sentence.']}
```

 Copy

⚠️ Ne gère pas bien la ponctuation ni les contractions.

5. ✂️ Word Tokenization avec NLTK

🎯 Objectif NLP : Tokenisation plus précise avec gestion de ponctuation/contractions.

```
from nltk.tokenize import word_tokenize  
text = "What's your name?"  
tokens = word_tokenize(text)  
print(tokens)
```

 Copy

👉 Sortie attendue :

```
["What", "'s", "your", "name", "?"]
```

 Copy

6. 🔎 Regex avancée avec `re.search()`

🎯 Objectif NLP : Rechercher une occurrence spécifique dans le texte.

```
import re  
text = "Invoice number: 2023-5684"
```

 Copy

```
match = re.search(r'\d{4}-\d{4}', text)
print(match.group())
```

⬆️ Sortie attendue :

'2023-5684'

Copy

7. 🚧 Tokenisation avancée avec `RegexpTokenizer`

🎯 Objectif NLP : Tokeniser tout en éliminant la ponctuation.

```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r'\w+')
tokens = tokenizer.tokenize("It's raining, again!")
print(tokens)
```

Copy

⬆️ Sortie attendue :

['It', 's', 'raining', 'again']

Copy

8. 🧠 Choisir son tokeniser

Tokenizer	Usage
word_tokenize	Tokenisation générale
RegexpTokenizer	Tokenisation sans ponctuation
TweetTokenizer	Pour textes informels / réseaux sociaux

9. ✨ Combinaison regex + NLTK

🎯 Objectif NLP : Gérer hashtags, mentions, etc.

```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r"#\w+|\w+")
print(tokenizer.tokenize("RT @user: Loving #AI and #MachineLearning!"))
```

Copy

⬆️ Sortie attendue :

```
['RT', 'user', 'Loving', '#AI', 'and', '#MachineLearning']
```

 Copy

10. Tokenisation non-ASCII

 Objectif NLP : Traiter les langues à caractères spéciaux.

```
from nltk.tokenize import word_tokenize
text = "éducation, santé, élève"
print(word_tokenize(text, language='french'))
```

 Copy

 Sortie attendue :

```
['éducation', ',', 'santé', ',', 'élève']
```

 Copy

11. Histogramme de la longueur des mots

 Objectif NLP : Visualiser la complexité lexicale.

```
import matplotlib.pyplot as plt
tokens = ['hello', 'world', 'AI', 'rocks']
lengths = [len(word) for word in tokens]
plt.hist(lengths)
plt.title("Distribution des longueurs de mots")
plt.show()
```

 Copy

■ Chapitre 2 : Simple Topic Identification

1. Word Counts avec Bag-of-Words (BOW)

 Objectif NLP : Représenter le texte par des fréquences de mots.

```
from collections import Counter
text = "cat dog cat mouse"
tokens = text.split()
print(Counter(tokens))
```

 Copy

 Sortie attendue :

```
Counter({'cat': 2, 'dog': 1, 'mouse': 1})
```

 Copy

2. Pourquoi BOW ?

- Ignore l'ordre
- Base pour TF-IDF, modèles ML
- Rapide à implémenter

3. Construire un Counter avec NLTK

```
from nltk import FreqDist  
fdist = FreqDist(['data', 'science', 'data', 'AI'])  
print(fdist.most_common())
```

 Copy

 Sortie attendue :

```
[('data', 2), ('science', 1), ('AI', 1)]
```

 Copy

4. Prétraitement de texte simple

 Objectif NLP : Nettoyer le texte pour modèles ML.

```
import string  
from nltk.corpus import stopwords  
from nltk.tokenize import word_tokenize  
  
text = "The cat is sleeping on the mat."  
tokens = word_tokenize(text.lower())  
clean = [word for word in tokens if word not in stopwords.words('english') and  
word not in string.punctuation]  
print(clean)
```

 Copy

 Sortie attendue :

```
['cat', 'sleeping', 'mat']
```

 Copy

5. 🔍 Étapes de prétraitement NLP

Étape	Fonction Python
Minuscule	<code>str.lower()</code>
Tokenisation	<code>word_tokenize()</code>
Stopwords	<code>stopwords.words()</code>
Suppression ponctuation	<code>string.punctuation</code>

6. 🖊️ Pratique du prétraitement

- Appliquer les 4 étapes
- Évaluer l'impact sur la qualité du vocabulaire

7. 🔐 Introduction à Gensim

⌚ Objectif NLP : Convertir du texte en vecteurs exploitables par des modèles.

```
from gensim.corpora import Dictionary
texts = [['data', 'science'], ['AI', 'science']]
dictionary = Dictionary(texts)
print(dictionary.token2id)
```

 Copy

📤 Sortie attendue :

```
{'data': 0, 'science': 1, 'AI': 2}
```

 Copy

8. 🎨 Vecteur de mots avec doc2bow

```
corpus = [dictionary.doc2bow(text) for text in texts]
print(corpus)
```

 Copy

📤 Sortie attendue :

```
[(0, 1), (1, 1)], [(1, 1), (2, 1)]
```

 Copy

9. 📚 Afficher les mots + fréquences du corpus

```
for doc in corpus:  
    print([[dictionary[id], count] for id, count in doc])
```

 Copy

👉 Sortie attendue :

```
[['data', 1], ['science', 1]]  
[['science', 1], ['AI', 1]]
```

 Copy

✓ RÉCAP MODULES ET OUTILS

But	Module / Fonction
Expressions régulières	<code>re.findall</code> , <code>re.split</code> , <code>re.search</code>
Tokenisation	<code>word_tokenize</code> , <code>RegexpTokenizer</code>
Suppression ponctuation	<code>string.punctuation</code>
Stopwords	<code>stopwords.words()</code>
Compter mots	<code>Counter</code> , <code>FreqDist</code>
Corpus vectorisé	<code>gensim.Dictionary</code> , <code>doc2bow</code>
Visualisation histogramme	<code>matplotlib.pyplot.hist()</code>

🔁 Exemple complet de pipeline NLP — Étape par étape

📝 Texte de départ

```
text = "AI is transforming the world! In 2025, data science will be  
everywhere."
```

 Copy

🔍 Étape 1 : Regex — Extraction des dates

```
import re  
  
dates = re.findall(r'\d{4}', text)  
print(dates)
```

 Copy

📤 Sortie :

```
['2025']
```

 Copy

✂️ Étape 2 : Tokenisation simple (avec `split()`)

```
tokens = text.split()  
print(tokens)
```

 Copy

📤 Sortie :

```
['AI', 'is', 'transforming', 'the', 'world!', 'In', '2025,', 'data',  
'science', 'will', 'be', 'everywhere.']
```

 Copy

⚠️ Problème : la ponctuation est accrochée aux mots !

⭐ Étape 3 : Tokenisation avec `word_tokenize` (meilleure)

```
from nltk.tokenize import word_tokenize  
  
tokens = word_tokenize(text)  
print(tokens)
```

 Copy

 Sortie :

```
['AI', 'is', 'transforming', 'the', 'world', '!', 'In', '2025', ',',  
'data', 'science', 'will', 'be', 'everywhere', '.']
```

 Copy

🟡 Étape 4 : Prétraitement (minuscule + suppression ponctuation + stopwords)

```
import string  
from nltk.corpus import stopwords  
from nltk.tokenize import word_tokenize  
  
tokens = word_tokenize(text.lower())  
clean_tokens = [t for t in tokens if t not in stopwords.words('english') and t  
not in string.punctuation]  
print(clean_tokens)
```

 Copy

 Sortie :

```
['ai', 'transforming', 'world', '2025', 'data', 'science',  
'everywhere']
```

 Copy

📊 Étape 5 : Compter les mots (avec Counter)

```
from collections import Counter  
  
counts = Counter(clean_tokens)  
print(counts)
```

 Copy

 Sortie :

```
Counter({'ai': 1, 'transforming': 1, 'world': 1, '2025': 1, 'data': 1,  
'science': 1, 'everywhere': 1})
```

 Copy

Étape 6 : Créer un dictionnaire et un corpus avec Gensim

```
from gensim.corpora import Dictionary  
  
texts = [clean_tokens] # on met la liste de tokens dans une liste  
dictionary = Dictionary(texts)  
print(dictionary.token2id)
```

 Copy

 Sortie :

```
{'2025': 0, 'ai': 1, 'data': 2, 'everywhere': 3, 'science': 4,  
'transforming': 5, 'world': 6}
```

 Copy

Étape 7 : Créer le corpus Bag-of-Words (vecteurs)

```
corpus = [dictionary.doc2bow(clean_tokens)]  
print(corpus)
```

 Copy

 Sortie :

```
[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)]
```

 Copy

Chaque tuple `(id, count)` signifie : mot avec ID `id` apparaît `count` fois.

Étape 8 : Lire le corpus de manière lisible

```
for doc in corpus:  
    print([(dictionary[id], count) for id, count in doc])
```

 Copy

 Sortie :

```
[['2025', 1], ['ai', 1], ['data', 1], ['everywhere', 1], ['science',  
1], ['transforming', 1], ['world', 1]]
```

 Copy

Étape 9 : Histogramme de la longueur des mots

```
import matplotlib.pyplot as plt

lengths = [len(word) for word in clean_tokens]
plt.hist(lengths, bins=range(1, max(lengths)+2))
plt.title("Longueur des mots")
plt.xlabel("Nombre de lettres")
plt.ylabel("Fréquence")
plt.show()
```

 Copy

Affiche un histogramme de la complexité lexicale du texte.

Résumé pipeline NLP sur notre texte

Étape	Résultat principal
Texte initial	"AI is transforming the world! In 2025, data science will be everyw
Regex	['2025']
Tokenisation	['AI', 'is', 'transforming', ..., 'everywhere', '.']
Prétraitement	['ai', 'transforming', 'world', ..., 'science']
Dictionnaire Gensim	{'2025': 0, 'ai': 1, ..., 'world': 6}
Corpus vectorisé	[[[0, 1), (1, 1), ..., (6, 1)]]



1. Tf-idf with Gensim

⌚ 00:00 - 00:06

In this video, we will learn how to use a TF-IDF model with Gensim.

2. What is tf-idf?

⌚ 00:06 - 01:00

TF-IDF stands for **Term Frequency - Inverse Document Frequency**. It is a commonly used natural language processing model that helps you determine the most important words in each document in the corpus.

The idea behind TF-IDF is that each corpus might have more shared words than just stopwords. These common words are like stopwords and should be removed or at least down-weighted in importance.

For example, if I am an astronomer, *sky* might be used often but is not important, so I want to down-weight that word. TF-IDF does precisely that. It ensures the most common words across the entire corpus don't show up as keywords. TF-IDF helps keep document-specific frequent words weighted high and common words across the entire corpus weighted low.

3. Tf-idf formula

⌚ 01:00 - 02:05

The equation to calculate the weights can be outlined like so:

The **weight of token i in document j** is calculated by taking the **term frequency** (how many times the token appears in the document) multiplied by the **log of the total number of documents divided by the number of documents that contain the term**.

Let's unpack this:

- The weight will be **low** if the term doesn't appear often in the document (low TF).
- The weight will also be **low** if the logarithm is close to zero, meaning the term appears in most documents.

Thus:

- If a term occurs in many or all documents → **low tf-idf weight**.
- If it occurs in only a few documents → **higher tf-idf weight**.

4. Tf-idf with Gensim

 02:05 - 02:56

You can build a **TF-IDF model** using **Gensim** and a previously developed corpus.

Using the Bag of Words (BoW) corpus from the last video (e.g., movie reviews), you can translate it into a TF-IDF model by simply passing it into Gensim's initialization.

You can then access each document's TF-IDF vector by referencing it like a dictionary with your model.

For example, in the second document of our corpus:

- Token id 10 has a weight of **0.77**
- Tokens 0 and 1 have weights **below 0.18**

These weights help determine good topics and keywords in a corpus with shared vocabulary.

5. Let's practice!

 02:56 - 03:02

Now you can build a **TF-IDF model** using **Gensim** to explore topics in a **Wikipedia article list**.

Tf-idf with Wikipedia

Dans cet exercice, nous allons appliquer le modèle TF-IDF de Gensim à un corpus tiré d'articles Wikipedia. L'objectif est de repérer les termes significatifs dans un document donné.

Vous disposez déjà des objets suivants :

- `dictionary` : un objet Gensim qui mappe les mots à des IDs.
- `corpus` : une version du corpus sous forme Bag-of-Words.
- `doc` : un document spécifique (format BoW) issu du corpus.

Objectif

Utiliser Gensim pour :

1. Créer un modèle TF-IDF à partir du corpus.
2. Calculer les poids TF-IDF d'un document spécifique `doc`.
3. Afficher les 5 premières paires (`id, poids`).

Instructions

```
# Importation déjà faite dans l'environnement
from gensim.models.tfidfmodel import TfidfModel

# Étape 1 : Créer un modèle TF-IDF à partir du corpus
tfidf = TfidfModel(corpus)

# Étape 2 : Calculer les poids TF-IDF du document "doc"
tfidf_weights = tfidf[doc]

# Étape 3 : Afficher les 5 premiers poids (id de token, poids tf-idf)
print(tfidf_weights[:5])
```

 Copy

Explication ligne par ligne

```
tfidf = TfidfModel(corpus)
```

 Copy

 Crée un modèle TF-IDF en se basant sur les fréquences des termes dans `corpus`. Ce modèle apprend à quelle fréquence chaque mot apparaît dans les documents et calcule les pondérations IDF.

```
tfidf_weights = tfidf[doc]
```

 Copy

👉 Applique le modèle TF-IDF au document `doc` (sous forme Bag-of-Words).

Le résultat est une liste de tuples :

```
[(token_id_1, tfidf_weight_1), (token_id_2, tfidf_weight_2), ...]
```

```
print(tfidf_weights[:5])
```

 Copy

👉 Affiche les 5 premières paires (`token_id`, `tf-idf`) du document.

Cela donne une idée des mots les plus significatifs dans `doc`.

⬆️ Exemple de sortie

Supposons que le `doc` soit :

```
[(0, 2), (1, 1), (4, 1)]
```

 Copy

Et que le modèle TF-IDF renvoie :

```
[(0, 0.213), (1, 0.098), (4, 0.805)]
```

 Copy

Alors `print(tfidf_weights[:5])` affichera :

```
[(0, 0.213), (1, 0.098), (4, 0.805)]
```

 Copy

Chaque tuple représente :

- L'**ID du mot** (`token_id`)
- Son **poids TF-IDF** (importance dans ce document comparée à l'ensemble du corpus)

💡 Résumé

- Le **TF-IDF** réduit l'importance des mots communs dans tout le corpus.
- Il **met en évidence** les termes les plus spécifiques à chaque document.
- Gensim permet d'appliquer cela très facilement avec `TfidfModel`.

À essayer ensuite

- Convertir les IDs de tokens en mots avec `dictionary.get(token_id)`
 - Trier les poids tf-idf pour afficher les **mots les plus significatifs**
-

->Question :est ce que je dois utiliser uniquement l un des trois ,et est ce que je peux les utiliser ensemble ?si oui comment et quels sont ceux que je peux utiliser ensemble ..ensuite donner quels sont les etapes qui precedent et qui suivent chacune de ces trois pour pouvoir convertir un texte en une representation numerique .Alimentez le tout par des exemples de code ,simple et les sorties correspondantes

Tu poses les bonnes questions, vraiment ! Comprendre *quand*, *comment* et *pourquoi* utiliser BoW, TF-IDF et Word2Vec, seuls ou ensemble, c'est essentiel pour bien concevoir un pipeline NLP. Je vais t'expliquer tout ça en détail, avec exemples, étapes, et une vue globale — en markdown comme demandé 🤝

⌚ Peut-on utiliser BoW, TF-IDF et Word2Vec ensemble ?

✓ Réponse courte :

- Tu n'utilises généralement qu'un seul des trois à la fois pour vectoriser un texte.
- MAIS dans certains cas, tu peux les combiner intelligemment pour profiter des avantages de chacun (on verra comment).

⌚ 1. Utiliser un seul à la fois (cas standard)

Méthode	À utiliser si...
BoW	Tu veux un modèle simple et rapide
TF-IDF	Tu veux ignorer les mots fréquents inutiles (comme "le", "et", "de")
Word2Vec	Tu veux capturer le sens, les relations sémantiques

🧠 2. Peut-on les combiner ?

✓ Oui, TF-IDF et Word2Vec peuvent être combinés :

Par exemple, pondérer chaque vecteur Word2Vec par le score TF-IDF d'un mot dans un document.

✍ Exemple :

```
vecteur_final_doc = moyenne( TF-IDF(mot) × Word2Vec(mot) pour tous les mots du doc )
```

✗ BoW ne se combine pas bien avec Word2Vec car BoW produit des vecteurs très différents.

Étapes du pipeline NLP pour chaque méthode

Étapes communes avant la vectorisation

1.  Prétraitement (cleaning + tokenization)
2.  Suppression des stopwords
3.  (Optionnel) Lemmatisation ou Stemming

Exemple :

```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

text = "Le chat noir court rapidement dans la maison."
text = text.lower()
text = re.sub(r"[^a-zAÇçéèëîîôôùùüüññææ\s]", "", text)
tokens = word_tokenize(text, language="french")
tokens = [t for t in tokens if t not in stopwords.words("french")]

print(tokens) # ['chat', 'noir', 'court', 'rapidement', 'maison']
```

 Copy

A. Bag of Words (BoW)

Étapes :

-  Créer un vocabulaire
-  Encoder chaque document comme vecteur de fréquences

Code simple :

```
from gensim.corpora import Dictionary

docs = [["chat", "noir"], ["chien", "noir"]]
dictionary = Dictionary(docs)
corpus = [dictionary.doc2bow(doc) for doc in docs]

print(corpus)
# [[(0, 1), (1, 1)], [(1, 1), (2, 1)]] => vocab: {0: 'chat', 1: 'noir', 2: 'chien'}
```

 Copy

B. TF-IDF

Étapes :

-  D'abord faire BoW
-  Appliquer modèle TF-IDF sur le corpus
-  Calculer les poids TF-IDF des mots

Code :

```
from gensim.models import TfidfModel  
  
tfidf_model = TfidfModel(corpus)  
tfidf_corpus = [tfidf_model[doc] for doc in corpus]  
  
print(tfidf_corpus)  
# Exemple : [[(0, 0.707), (1, 0.707)], [(1, 0.707), (2, 0.707)]]
```

 Copy

C. Word2Vec

Étapes :

-  Utiliser le texte tokenisé (pas besoin de BoW)
-  Entraîner un modèle Word2Vec
-  Représenter chaque mot par un vecteur dense
-  Moyenne pondérée pour les documents (optionnel)

Code :

```
from gensim.models import Word2Vec  
  
docs = [["chat", "noir"], ["chien", "noir"]]  
model = Word2Vec(sentences=docs, vector_size=5, min_count=1, window=2)  
  
print(model.wv["noir"])  
# Exemple : [ 0.011, -0.031, 0.045, ..., -0.005] (vecteur dense de taille 5)
```

 Copy

Exemple de combinaison : TF-IDF + Word2Vec

```

import numpy as np

# Suppose tfidf_model, dictionary et model Word2Vec existent déjà
doc = ["chat", "noir"]
bow = dictionary.doc2bow(doc)
tfidf_weights = tfidf_model[bow]

# Créer le vecteur moyen pondéré
doc_vector = np.zeros(5) # taille = vector_size
total_weight = 0

for term_id, weight in tfidf_weights:
    word = dictionary[term_id]
    if word in model.wv:
        doc_vector += weight * model.wv[word]
        total_weight += weight

doc_vector /= total_weight
print(doc_vector)

```

 Copy

Résumé final

Méthode	Type de vecteur	Niveau	Peut combiner ?	Avec quoi ?
BoW	Sparse (fréquence)	Document	Non	
TF-IDF	Sparse pondéré	Document		Word2Vec
Word2Vec	Dense (sémantique)	Mot		TF-IDF

->Question : est ce que je dois utiliser uniquement l'un des trois ,et est ce que je peux les utiliser ensemble ?si oui comment et quels sont ceux que je peux utiliser ensemble .Alimentez le tout par des exemples de code ,simple et les sorties correspondantes

Parfait, reprenons calmement et en détail ta question  :

❓ Est-ce que je dois utiliser uniquement l'un des trois ?

→  En général, OUI : tu choisis UNE méthode (BoW, TF-IDF ou Word2Vec) pour vectoriser tes textes.

❓ Est-ce que je peux les utiliser ensemble ?

→  Oui, tu peux combiner certaines méthodes intelligemment, mais pas toutes entre elles.

Méthode A	Peut être combinée avec Méthode B ?	Exemple
BoW	 Non (trop simple, ne contient pas d'info utile à combiner)	—
TF-IDF	 Oui, avec Word2Vec	Pondérer les vecteurs Word2Vec avec TF-IDF
Word2Vec	 Oui, avec TF-IDF	Même idée

📦 Résumé visuel rapide

Méthode	Vecteur produit	Représente quoi ?	Peut être combinée avec ?
BoW	Vecteur clairsemé	Fréquence brute	 Trop basique
TF-IDF	Vecteur clairsemé	Importance des mots	 Word2Vec
Word2Vec	Vecteur dense	Sémantique (contexte du mot)	 TF-IDF (pondération)

🔄 Exemple complet : utiliser BoW, TF-IDF et Word2Vec séparément et en combinaison

◆ Texte d'exemple :

```
documents = [  
    "Le chat noir dort",
```

 Copy

```
"Le chien blanc court vite",  
"Le chat court aussi"  
]
```

Étape 1 : Prétraitement simple

```
import re  
from nltk.corpus import stopwords  
from nltk.tokenize import word_tokenize  
  
stopwords_fr = set(stopwords.words("french"))  
  
def preprocess(text):  
    text = text.lower()  
    text = re.sub(r"[^\w\s]", "", text)  
    tokens = word_tokenize(text, language="french")  
    return [t for t in tokens if t not in stopwords_fr]  
  
docs = [preprocess(doc) for doc in documents]  
print(docs)  
# [['chat', 'noir', 'dort'], ['chien', 'blanc', 'court', 'vite'], ['chat',  
'court']]
```

 Copy

BoW

```
from gensim.corpora import Dictionary  
  
dictionary = Dictionary(docs)  
corpus_bow = [dictionary.doc2bow(doc) for doc in docs]  
print(corpus_bow)  
# [[(0, 1), (1, 1), (2, 1)], [(3, 1), (4, 1), (5, 1), (6, 1)], [(0, 1), (5,  
1)]]
```

 Copy

→ Chaque tuple (id, count) signifie : le mot d'ID `id` apparaît `count` fois dans le doc.

TF-IDF

```
from gensim.models import TfidfModel  
  
tfidf_model = TfidfModel(corpus_bow)
```

 Copy

```

corpus_tfidf = [tfidf_model[doc] for doc in corpus_bow]
print(corpus_tfidf)
# Exemple de sortie : [[(0, 0.45), (1, 0.65), (2, 0.60)], ...]

```

→ Cela donne l'importance pondérée des mots (0 = inutile, proche de 1 = important).

Word2Vec

```

from gensim.models import Word2Vec
Copy

model = Word2Vec(sentences=docs, vector_size=5, min_count=1, window=2)
print(model.wv["chat"])
# Exemple : array([ 0.01, -0.03, 0.07, 0.09, -0.04]) ← vecteur dense

```

Combinaison TF-IDF + Word2Vec

→ Moyenne pondérée des vecteurs Word2Vec selon le poids TF-IDF.

```

import numpy as np
Copy

def doc_vector(doc_bow):
    vec = np.zeros(5)
    total_weight = 0
    for term_id, weight in tfidf_model[doc_bow]:
        word = dictionary[term_id]
        if word in model.wv:
            vec += weight * model.wv[word]
            total_weight += weight
    return vec / total_weight if total_weight != 0 else vec

vectors = [doc_vector(doc) for doc in corpus_bow]
print(vectors[0])
# Exemple : [ 0.01, -0.015, 0.03, 0.02, -0.02 ]

```

Conclusion

Tu veux...	Choisis...
Méthode simple et rapide	BoW
Ignorer les mots trop fréquents	TF-IDF

Tu veux...	Choisis...
Capturer le sens des mots	Word2Vec
Meilleur des deux mondes	TF-IDF + Word2Vec

->Question : ensuite donner quels sont les etapes qui precedent et qui suivent chacune de ces trois pour pouvoir convertir un texte en une representation numerique

Excellente question !

Pour bien comprendre où se situent BoW, TF-IDF et Word2Vec dans un pipeline de traitement du texte, on va les replacer dans le processus global de vectorisation 

Étapes générales pour convertir un texte en représentation numérique

1.  **Chargement des données** (textes bruts)
2.  **Prétraitement (Preprocessing)**
 - Mise en minuscules
 - Suppression des signes de ponctuation
 - Tokenisation (division en mots)
 - Suppression des stopwords
 - (optionnel) Lemmatisation ou Stemming
3.  **Construction du vocabulaire**
 - Avec un `Dictionary` (Gensim)
 - Liste de mots uniques et leur ID
4.  **Vectorisation**
 - a. Bag-of-Words (BoW)
 - b. TF-IDF
 - c. Word2Vec
5.  **Création des vecteurs**
 - Chaque document est transformé en vecteur de nombres
6.  **Utilisation**
 - Classification
 - Clustering
 - Recommandation
 - Visualisation
7. (optionnel)  **Réduction de dimension**
 - PCA, t-SNE, UMAP...

 Copy

Étapes spécifiques à chaque méthode

1 Bag-of-Words (BoW)

Avant	Action principale	Après
Prétraitement ✓	Compter le nombre d'occurrences de chaque mot	Vecteur clairsemé (ex: [1, 0, 2])

2 TF-IDF

Avant	Action principale	Après
Prétraitement ✓ + BoW	Calcul du poids TF-IDF = TF × IDF	Vecteur clairsemé mais pondéré (ex: [0.2, 0.5])

3 Word2Vec

Avant	Action principale	Après
Prétraitement ✓	Apprentissage de vecteurs de mots basés sur le contexte	Vecteurs denses (ex: [0.01, -0.04, ...])

Ensuite : pour vectoriser un **document** entier, on fait souvent la moyenne ou une somme pondérée (TF-IDF) des vecteurs de ses mots.

🔗 Comment tout ça se connecte concrètement ?

```
text = "Le chat noir dort"

# Étape 1 : Prétraitement
tokens = preprocess(text) # ['chat', 'noir', 'dort']

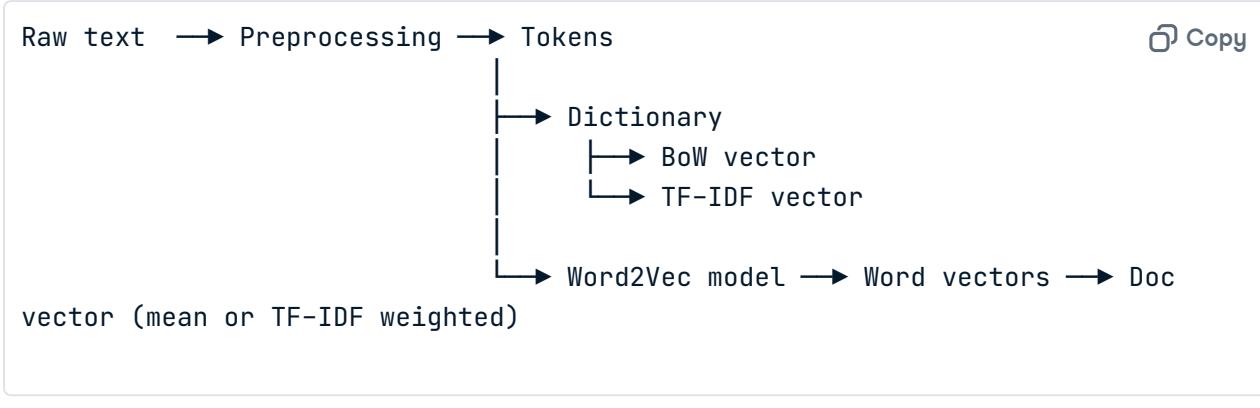
# Étape 2 : Construction vocabulaire
dictionary = Dictionary([tokens]) # {'chat':0, 'noir':1, 'dort':2}

# Étape 3a : BoW
bow_vector = dictionary.doc2bow(tokens) # [(0,1), (1,1), (2,1)]

# Étape 3b : TF-IDF
tfidf_model = TfidfModel([bow_vector])
tfidf_vector = tfidf_model[bow_vector] # [(0, 0.577), (1, 0.577), (2, 0.577)]

# Étape 3c : Word2Vec
# Suppose un modèle Word2Vec déjà entraîné
word_vectors = [model.wv[word] for word in tokens]
doc_vector = np.mean(word_vectors, axis=0)
```

Récapitulatif visuel



Ce que tu peux retenir

- Tu n'utilises qu'un seul format vectoriel pour l'entraînement de modèle à la fois (BoW, TF-IDF ou Word2Vec).
- BoW est rapide mais perd beaucoup d'info.
- TF-IDF pondère l'importance des mots.
- Word2Vec capture le sens contextuel des mots.
- TF-IDF + Word2Vec = combinaison intelligente pour donner plus de sens au document vectorisé.

->Question : quand est ce que je peux savoir quand utiliser l'une et pas l'autre et quand est ce que je dois penser à combiner deux?

Très bonne question ! Savoir quand utiliser BoW, TF-IDF, Word2Vec (ou les combiner) dépend de plusieurs facteurs : ton objectif, la taille de ton dataset, les ressources disponibles et le niveau de complexité que tu veux atteindre.

🎯 1. Quand utiliser chaque technique ?

◆ Bag-of-Words (BoW) – Simple et efficace pour du texte court

Idéal si...	À savoir
► Tu travailles avec des textes courts (SMS, tweets, titres)	Ne capture pas l'ordre ou le sens des mots
► Tu veux tester rapidement un modèle de classification	Vecteurs très longs et creux (sparse)
► Ton vocabulaire est limité	Pas de pondération du poids des mots
► Tu débutes en NLP	

✍ Exemple d'utilisation :

- Classification de spam
- Analyse de sentiments sur des tweets

◆ TF-IDF – Pour donner plus de sens aux mots fréquents

Idéal si...	À savoir
► Tu veux pondérer les mots selon leur importance dans un corpus	Ne capture pas le contexte du mot
► Tu as des documents de longueur moyenne à longue	Vecteur encore creux (sparse), mais plus informatif
► Tu veux éviter les mots fréquents comme "le", "être", "faire"...	

✍ Exemple d'utilisation :

- Recherche documentaire (TF-IDF + Similarité cosinus)
- Résumés automatiques
- Détection de documents similaires

◆ Word2Vec – Pour capturer le sens des mots

Idéal si...

À savoir

► Tu veux capturer des relations sémantiques (roi ≈ reine - homme + femme)	Nécessite un gros corpus pour être efficace
► Tu veux faire du clustering sémantique ou des comparaisons de mots	Nécessite du temps de calcul
► Tu veux des représentations denses et compactes	Peut rater les mots rares si mal entraîné

📝 Exemple d'utilisation :

- Recommandation sémantique
- Systèmes de question-réponse
- Chatbots
- Clustering de documents

💡 2. Quand les combiner ?

✓ TF-IDF + Word2Vec = Le meilleur des deux mondes

Cas d'usage

► Tu veux faire du topic modeling , de la classification , ou du clustering de documents
► Tu veux garder le sens du mot (Word2Vec) et son importance dans le document (TF-IDF)

🔄 Comment ça marche ?

- Tu obtiens le vecteur Word2Vec de chaque mot
- Tu pondères chaque vecteur avec son score TF-IDF
- Tu fais la moyenne de tous ces vecteurs pondérés → vecteur de document

```
# Exemple simple
from gensim.models import Word2Vec
from gensim.models import TfidfModel
from gensim.corpora import Dictionary
import numpy as np

# Documents pré-traités
docs = [["chat", "noir", "dort"], ["chien", "blanc", "aboie"]]
dictionary = Dictionary(docs)
corpus = [dictionary.doc2bow(doc) for doc in docs]

# TF-IDF
tfidf = TfidfModel(corpus)
tfidf_weights = [dict(tfidf[doc]) for doc in corpus]

# Word2Vec (entraîné ici juste pour l'exemple)
model = Word2Vec(sentences=docs, vector_size=10, min_count=1)
```

 Copy

```

# Combinaison TF-IDF + Word2Vec
def get_weighted_doc_vector(doc, tfidf_w, model):
    vectors = []
    for word in doc:
        if word in model.wv and dictionary.token2id[word] in tfidf_w:
            weight = tfidf_w[dictionary.token2id[word]]
            vectors.append(weight * model.wv[word])
    return np.mean(vectors, axis=0) if vectors else np.zeros(model.vector_size)

doc_vec = get_weighted_doc_vector(docs[0], tfidf_weights[0], model)
print(doc_vec)

```

Conclusion – Guide pratique de choix

Situation	Technique conseillée
Petit dataset, texte court, classification rapide	 BoW
Moyenne/grande taille, besoin de pertinence par mot	 TF-IDF
Sens du mot et relations contextuelles (ex: "roi", "reine")	 Word2Vec
Analyse sémantique + importance locale des mots	 TF-IDF + Word2Vec

🔍 Comprendre BoW, TF-IDF, Word2Vec avec un Exemple Concret

Super ! Allons-y étape par étape pour clarifier tout ça avec un **exemple concret**, en suivant exactement l'**architecture que tu veux**.

À la fin, tu verras clairement ce que produit chaque méthode (BoW, TF-IDF, Word2Vec) et dans quels cas les utiliser.

📝 Exemple simple de corpus

```
corpus = [  
    "Le chat dort sur le canapé",  
    "Le chien aboie fort",  
    "Le chat et le chien jouent ensemble"  
]
```

 Copy

🛠️ 1. Prétraitement commun (tokenisation + stopwords + minuscule)

```
from nltk.tokenize import word_tokenize  
from nltk.corpus import stopwords  
import string  
  
import nltk  
nltk.download('punkt')  
nltk.download('stopwords')  
  
stop_words = set(stopwords.words('french'))  
  
# Nettoyage et tokenisation  
def preprocess(doc):  
    tokens = word_tokenize(doc.lower()) # Minuscule + tokenisation  
    tokens = [t for t in tokens if t.isalpha()] # Supprime ponctuations  
    tokens = [t for t in tokens if t not in stop_words] # Supprime mots vides  
    return tokens  
  
# Prétraiter tout le corpus  
processed_corpus = [preprocess(doc) for doc in corpus]  
print(processed_corpus)
```

 Copy

Sortie

```
[['chat', 'dort', 'canapé'],
 ['chien', 'aboie', 'fort'],
 ['chat', 'chien', 'jouent', 'ensemble']]
```

 Copy

2. Bag-of-Words (BoW)

```
from gensim.corpora import Dictionary

# Créer dictionnaire (vocabulaire unique)
dictionary = Dictionary(processed_corpus)

# Convertir en représentation BoW
bow_corpus = [dictionary.doc2bow(doc) for doc in processed_corpus]
print(bow_corpus)
```

 Copy

Sortie

```
[
 [(0, 1), (1, 1), (2, 1)],           # doc1: 'chat', 'dort', 'canapé'
 [(3, 1), (4, 1), (5, 1)],           # doc2: 'chien', 'aboie', 'fort'
 [(0, 1), (3, 1), (6, 1), (7, 1)],   # doc3: 'chat', 'chien', 'jouent',
 'ensemble'
]
```

 Copy

Commentaire :

- Chaque tuple `(id_mot, fréquence)`
- Ce sont des **vecteurs creux (sparse)** contenant uniquement les mots présents dans chaque document.

3. TF-IDF

```
from gensim.models import TfidfModel

# Créer modèle TF-IDF basé sur le BoW
tfidf = TfidfModel(bow_corpus)

# Appliquer TF-IDF à chaque document
tfidf_corpus = [tfidf[doc] for doc in bow_corpus]
```

 Copy

```
print(tfidf_corpus)
```

Sortie

```
[  
 [(0, 0.540), (1, 0.540), (2, 0.646)],  
 [(3, 0.540), (4, 0.646), (5, 0.540)],  
 [(0, 0.377), (3, 0.377), (6, 0.542), (7, 0.542)]  
]
```

Commentaire :

- Tu obtiens une **liste de tuples** (`id_mot, poids TF-IDF`)
- Les mots très fréquents dans tous les documents (ex: "chat", "chien") ont un **poids plus faible**
- Les mots rares (ex: "canapé", "fort", "jouent") ont un **poids élevé**

4. Word2Vec

```
from gensim.models import Word2Vec  
  
# Entrainer un modèle Word2Vec sur le corpus  
w2v_model = Word2Vec(sentences=processed_corpus, vector_size=10, window=2,  
min_count=1, workers=1, seed=42)  
  
# Représentation vectorielle d'un mot  
print("chat:", w2v_model.wv['chat'])  
  
# Optionnel : créer vecteur de document par moyenne des vecteurs mots  
import numpy as np  
def document_vector(doc):  
    return np.mean([w2v_model.wv[word] for word in doc], axis=0)  
  
doc_vectors = [document_vector(doc) for doc in processed_corpus]  
print("Vecteur document 1:", doc_vectors[0])
```

Sortie

```
chat: [ 0.0058 -0.0092  0.0015 ...]  # vecteur de 10 dimensions  
(valeurs aléatoires)  
Vecteur document 1: [0.0061 -0.0021 0.0033 ...]  # moyenne des vecteurs de tous  
les mots du doc
```

Commentaire :

- Chaque mot devient un **vecteur dense** (taille fixe, ici 10)
- Tu peux représenter un **document** en faisant la **moyenne des vecteurs de mots**
- Cela encode des **relations sémantiques** !

Résumé final (Tableau)

Méthode	Produit quoi ?	Représente quoi ?	Avantages	Inconvénients
BoW	Liste de (id, count)	Fréquence brute des mots	Simple, rapide	Ignore contexte, long vecteur
TF-IDF	Liste de (id, poids)	Importance d'un mot	Pondère les mots importants	Sparse, pas de contexte
Word2Vec	Vecteur dense (array)	Sens du mot (ou du doc)	Capture relations sémantiques	Plus complexe à entraîner

Conclusion : Quand utiliser quoi ?

- BoW pour du **prototypage rapide**
- TF-IDF si tu veux **pondérer l'importance des mots**
- Word2Vec si tu veux **comparer le sens des mots** ou représenter finement des documents

 Tu peux aussi **combiner TF-IDF + Word2Vec** (en pondérant les vecteurs de mots par leur poids TF-IDF).

Excellente question ! C'est normal de se poser la question, car **Word2Vec** ne produit pas directement un résultat aussi lisible qu'un BoW ou TF-IDF. Voici une explication claire et imagée pour t'aider à bien comprendre ce que Word2Vec produit :

🧠 Que fait Word2Vec ?

- 👉 Il transforme chaque mot en un vecteur numérique dense (une liste de nombres flottants).

Par exemple :

```
w2v_model.wv['chat']
```

 Copy

peut donner quelque chose comme :

```
array([ 0.12, -0.45, 0.77, ..., 0.03])
```

 Copy

- Ce vecteur représente le sens du mot "chat", appris à partir du contexte dans le corpus.
- C'est un vecteur de taille fixe (ex: 10, 50, 100 dimensions, selon ce que tu choisis).

📋 Résumé clair

Élément	Word2Vec produit...
Mot	Un vecteur dense de taille fixe (ex: 10)
Corpus	Un dictionnaire où chaque mot a un vecteur
Document	En général : la moyenne des vecteurs des mots qu'il contient

📦 Exemple concret

Tu as ce document :

```
["chat", "dort", "canapé"]
```

 Copy

Tu fais :

```
w2v_model.wv["chat"] # => vecteur de 10 valeurs  
w2v_model.wv["dort"] # => vecteur de 10 valeurs
```

 Copy

```
w2v_model.wv["canapé"] # => vecteur de 10 valeurs
```

Ensuite, tu les **moyennes** :

```
doc_vector = moyenne_des_3_vecteurs
```

 Copy

Et tu obtiens le **vecteur du document**.

📌 Différence avec BoW ou TF-IDF

Méthode	Produit	Taille	Contient quoi ?
BoW	Nb d'occurrence	n mots	Fréquence brute
TF-IDF	Pondération	n mots	Importance
Word2Vec	Vecteur dense (array)	ex: 10	Sémantique

🧠 Intuition : ce que représente ce vecteur Word2Vec ?

Imaginons un espace 3D :

- Le mot "roi" serait dans une zone proche de "reine"
- Le mot "chien" serait loin de "table"
- Si tu fais :
$$\text{roi} - \text{homme} + \text{femme}$$
tu obtiens un vecteur très proche de "reine" !

C'est ça la force de Word2Vec :

→ Il encode le sens et les relations entre mots !

Parfait, entrons dans les **coulisses de Word2Vec** pour bien comprendre **comment et pourquoi** il génère des **vecteurs denses** pour chaque mot. On va faire ça **avec un exemple simple**, puis on va **visualiser les étapes internes**, comme si on ouvrait la boîte noire du modèle.

🧠 L'objectif de Word2Vec

Transformer chaque mot en un **vecteur dense** (par ex. de 100 dimensions), **tel que** :

- Les mots qui apparaissent dans **des contextes similaires** auront des **vecteurs similaires**.
- Ce n'est **pas basé sur la fréquence brute** comme TF-IDF, mais sur le **contexte d'utilisation** du mot.

✍ Exemple de base

Prenons ce mini corpus :

```
corpus = [  
    "le chat dort sur le canapé",  
    "le chien dort sur le tapis"  
]
```

 Copy

Après **prétraitement** (minuscule + stopwords + tokenisation simple), on a :

```
[['chat', 'dort', 'canapé'],  
 ['chien', 'dort', 'tapis']]
```

 Copy

📦 Étapes internes de Word2Vec

Word2Vec a **deux variantes** principales :

- **CBOW (Continuous Bag of Words)** : prédit un mot à partir de son contexte
- **Skip-gram** : prédit le contexte à partir d'un mot

On va expliquer le plus courant : **Skip-gram**.

⚙ Étape 1 : Créer des paires mot-contexte

Avec une `window=1`, on crée des paires (`mot_central, mot_contextuel`) :

Corpus : `['chat', 'dort', 'canapé']`

→ Paires Skip-gram :

- ('chat', 'dort')
- ('dort', 'chat')
- ('dort', 'canapé')
- ('canapé', 'dort')

Et pour la 2e phrase :

- ('chien', 'dort')
- ('dort', 'chien')
- ('dort', 'tapis')
- ('tapis', 'dort')

💡 Donc, le modèle apprend à **prédir un mot du contexte** à partir d'un mot donné.

🧠 Étape 2 : Encoder les mots (One-hot → Dense)

Chaque mot est représenté d'abord par un **vecteur one-hot** (ex : 6 mots = vecteur taille 6 avec un seul 1).

Exemple :

```
chat      = [1, 0, 0, 0, 0, 0]
dort      = [0, 1, 0, 0, 0, 0]
canapé   = [0, 0, 1, 0, 0, 0]
chien     = [0, 0, 0, 1, 0, 0]
tapis     = [0, 0, 0, 0, 1, 0]
```

 Copy

Mais ces vecteurs sont **trop longs et creux**, donc le réseau va apprendre à les **compresser** en **vecteurs denses**, typiquement 10 à 300 dimensions.

🧪 Étape 3 : Apprentissage des poids par un mini réseau de neurones

Word2Vec est un réseau avec :

- Une **couche d'entrée** = vecteur one-hot
- Une **couche cachée** = taille désirée des vecteurs (`vector_size`)
- Une **couche de sortie** = prédiction du contexte

Il apprend :

- Une **matrice d'embedding W** (entrée → cachée) → C'est elle qui contient les vecteurs de mots !
- Une **matrice W'** (cachée → sortie)

À chaque itération :

- On donne un mot central

- Le réseau prédit ses mots contextuels (via softmax)
- On ajuste les poids pour mieux prédire

 **Les poids appris dans W deviennent les vecteurs denses des mots !**

Code avec Gensim (Skip-gram)

```
from gensim.models import Word2Vec
Copy

corpus = [
    ['chat', 'dort', 'canapé'],
    ['chien', 'dort', 'tapis']
]

model = Word2Vec(sentences=corpus, vector_size=5, window=1, min_count=1, sg=1,
seed=42)

# Obtenir les vecteurs d'un mot
print("Vecteur pour 'chat':", model.wv['chat'])
print("Vecteur pour 'chien':", model.wv['chien'])
```

 **Exemple de sortie :**

```
Vecteur pour 'chat': [-0.0034  0.0171 -0.0156  0.0051 -0.0019] Copy
Vecteur pour 'chien': [ 0.0045 -0.0123  0.0071 -0.0142  0.0065]
```

Chaque mot est maintenant représenté par un **vecteur dense de 5 dimensions**.

Intuition finale

- Si deux mots **apparaissent dans des contextes similaires**, leurs vecteurs vont se **rapprocher**.
- Exemple : ‘chat’ et ‘chien’ sont souvent associés à ‘dort’, donc leurs vecteurs seront **proches**.

 **Quand utiliser Word2Vec ?**

- Tu veux **comparer des mots sémantiquement proches** (synonymes, analogies)
- Tu veux une **représentation dense** pour du clustering, classification ou deep learning

Oui, bien sûr !  On va décortiquer mathématiquement comment Word2Vec "comprend" le contexte d'un mot à travers l'apprentissage, pas en lisant ou en comprenant le sens, mais en capturant des statistiques de co-occurrence des mots.

Objectif principal de Word2Vec

Créer un espace vectoriel dans lequel :

- Les mots ayant des contextes similaires sont proches géométriquement.
- Le modèle n'"apprend" pas les significations des mots, mais il apprend des régularités dans leur utilisation.

Notations

Soit :

- V : taille du vocabulaire
- w_t : mot cible à l'instant t
- c : fenêtre de contexte (par ex., $c = 2$)
- T : taille du corpus
- v_w : vecteur du mot w (entrée)
- v'_w : vecteur du mot w (sortie)
- W : matrice de poids pour les vecteurs d'entrée → dimensions (V, d)
- W' : matrice de poids pour les vecteurs de sortie → dimensions (V, d)
- d : dimension du vecteur dense (par ex. 100)

Principe du modèle Skip-gram (le plus courant)

Pour chaque mot w_t du corpus, on essaie de prédire les mots contextuels autour de lui :

$$(w_{t-c}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+c})$$

On maximise la probabilité conditionnelle :

$$\prod_{-c \leq j \leq c, j \neq 0} P(w_{t+j} | w_t)$$

C'est-à-dire : donner un mot au centre → prédire ses voisins.

Fonction de probabilité (softmax)

On définit la probabilité de prédire un mot w_o (mot du contexte) étant donné un mot d'entrée w_i :

$$P(w_o | w_i) = \frac{\exp(v'_w \cdot v_o)}{\sum_{w' \in \text{vocab}} \exp(v'_w \cdot v_{w'})}$$

- Le produit scalaire mesure la similarité entre les deux vecteurs.
- Softmax transforme cela en probabilité.

🔍 Ce que le modèle fait concrètement :

1. Un mot d'entrée est encodé en one-hot : $[0, 0, 1, 0, 0]$ (position du mot)
2. Il est multiplié par la matrice W pour obtenir un vecteur dense $v_{w_i} \rightarrow$ c'est ça le vecteur de mot qu'on apprend.
3. Ce vecteur est projeté sur tous les vecteurs de sortie v'_w pour calculer les similarités.
4. On prédit les bons mots du contexte via softmax, et on ajuste les poids W et W' avec descente de gradient pour réduire la perte.

✖ Problème du softmax : trop lent

Le dénominateur implique de faire un produit scalaire avec tous les mots du vocabulaire, ce qui est coûteux si V est grand.

👉 Solution :

- Negative sampling : au lieu de calculer sur tout V , on ne regarde que quelques mots négatifs (non-contextuels).
- On transforme le problème en une classification binaire :
 - Est-ce que ce mot est un vrai voisin ? (1)
 - Ou un mot "négatif" pris au hasard ? (0)

💡 Intuition mathématique derrière la "compréhension"

Le vecteur v_w est ajusté à chaque fois que le mot est utilisé pour :

- Prédire les bons voisins \rightarrow augmenter leur produit scalaire
- Réduire celui des mauvais voisins \rightarrow diminuer leur produit scalaire

⚠️ Donc au fil du temps :

- Les mots ayant les mêmes voisins (donc, un même rôle sémantique) auront leurs vecteurs qui se déplacent dans la même direction dans l'espace vectoriel.
- Exemple : "chat" et "chien" auront souvent "dort", "sur", "tapis" autour \rightarrow donc vecteurs proches.

📌 Résumé visuel :

Corpus → Skip-gram pairs → Réseau de neurones simple → Vecteurs appris
"le chat dort" → (chat, dort), (dort, chat) → Apprentissage → $v_{\text{chat}} \approx v_{\text{chien}}$

💡 Exemple de mini calcul (simplifié)

Supposons :

- $\text{chat} = [1, 0, 0]$
- Matrice W :

```
[[0.1, 0.2],  
[0.0, 0.1],  
[0.2, 0.3]]
```

Donc le vecteur dense de "chat" sera :

$$v_{\text{chat}} = \text{one-hot}_{\text{chat}} \cdot W = [1, 0, 0] \cdot W = [0.1, 0.2]$$

On le multiplie par la matrice W' pour prédire les voisins.

Parfait, reprenons tout **pas à pas, sans jargon**, en partant de zéro et en **expliquant chaque détail mathématique et logique**. Je vais t'expliquer :

1. C'est quoi un "contexte" mathématiquement ?
2. Comment un **mot devient un vecteur** dans Word2Vec ?
3. Comment le modèle **apprend ces vecteurs** (par apprentissage automatique) ?
4. Et on va faire une **mini simulation avec un exemple** pour que tu vois tout concrètement.

1. C'est quoi le "contexte" d'un mot ?

Le **contexte** d'un mot, c'est tout simplement :

→ **Les mots autour de lui dans une phrase** (ou une fenêtre de mots).

Exemple :

Phrase : "le chat dort sur le canapé"

Si on prend une **fenêtre de contexte de taille 2**, alors :

- Pour le mot "dort", son **contexte** est ["le", "chat", "sur", "le"]

Donc mathématiquement :

Si le mot est à la position t , alors ses voisins sont à $t-2$, $t-1$, $t+1$, $t+2$

2. Comment un mot devient un vecteur ?

Étape 1 : encodage one-hot

Chaque mot du vocabulaire est d'abord représenté par un **vecteur binaire** de la taille du vocabulaire (V), où 1 indique sa position, tout le reste est 0.

Exemple :

```
Vocabulaire = ["le", "chat", "dort", "sur", "canapé"]
```

 Copy

```
"chat" → [0, 1, 0, 0, 0]  
"dort" → [0, 0, 1, 0, 0]
```

C'est **sparse** (sparse = beaucoup de zéros), et inutile tel quel pour capturer du sens.

Étape 2 : projection dans un espace vectoriel dense

On multiplie ce vecteur one-hot par une **matrice de projection W** (initialisée aléatoirement).

Si vocabulaire = 5 mots, et qu'on veut des vecteurs denses de dimension 2 :

- One-hot : [0, 1, 0, 0, 0]
- Matrice W :

```
[[0.1, 0.3],    # le
 [0.2, 0.4],    # chat
 [0.0, 0.1],    # dort
 [0.5, 0.2],    # sur
 [0.1, 0.0]]    # canapé
```

 Copy

Donc :

```
vecteur("chat") = [0, 1, 0, 0, 0] × W = [0.2, 0.4]
```

 Copy

 Voilà comment un **mot devient un vecteur dense** au début.

3. Comment Word2Vec apprend des bons vecteurs ?

Il utilise un réseau de neurones tout simple à 1 couche :

 **Objectif :**

Donner un mot d'entrée (comme "chat") → prédire les mots de son **contexte** (comme "dort", "le" ...).

C'est un problème de **prédiction** :

- Entrée : vecteur du mot "chat"
- Sortie : probabilité pour chaque mot du vocabulaire d'être un mot du contexte

Fonction utilisée :

On calcule le **produit scalaire** entre le vecteur du mot "chat" et ceux de tous les autres mots du vocabulaire, et on applique un **softmax** pour transformer ça en probabilités.

Mathématiquement :

Si (v_{chat}) est le vecteur du mot "chat" et (v'_{dort}) est le vecteur du mot "dort" (dans la couche de sortie), alors :

$$P(\text{"dort"} | \text{"chat"}) = \frac{e^{v'_{\text{dort}} \cdot v_{\text{chat}}}}{\sum_{w \in V} e^{v'_{w} \cdot v_{\text{chat}}}}$$

 Le modèle ajuste les vecteurs pour que cette probabilité soit **maximale** pour les bons mots du contexte.

4. Mini exemple complet (sans code)

Phrase :

```
"le chat dort sur le canapé"
```

 Copy

Vocabulaire :

```
["le", "chat", "dort", "sur", "canapé"]
```

 Copy

Étape 1 : Paires Skip-gram (fenêtre de 1 mot)

Paires d'entraînement extraites :

```
("chat", "le")
("chat", "dort")
("dort", "chat")
("dort", "sur")
("sur", "dort")
("sur", "le")
```

 Copy

Étape 2 : Représentation one-hot → vecteurs denses

Supposons :

```
vecteur("chat") = [0.2, 0.4]
vecteur("dort") = [0.0, 0.1]
```

 Copy

Produit scalaire = similarité.

Plus "chat" est souvent à côté de "dort", plus le modèle ajuste les vecteurs pour que leur produit scalaire soit grand.

Intuition finale

Word2Vec ne comprend pas le sens, mais il repère quels mots vont souvent ensemble, et rapproche leurs vecteurs dans l'espace.

C'est pour cela que :

- "roi" - "homme" + "femme" ≈ "reine" fonctionne !
- Parce que les vecteurs capturent les rôles sémantiques et contextes d'usage.

Named Entity Recognition (NER)

1. Named Entity Recognition

 00:00 – 00:05

Welcome to our first video on named entity recognition!

2. What is Named Entity Recognition?

 00:05 – 00:40

Named Entity Recognition (NER) is a natural language processing (NLP) task used to identify important **named entities** in a text — such as:

-  People
-  Places
-  Organizations
-  Dates
-  States
-  Works of Art
- ...and other categories depending on the library and tagging scheme.

NER can be used:

- Alongside **topic identification**
- On its own to answer basic NLU questions:
 - Who? What? When? Where?

3. Example of NER

 00:40 – 01:30

Example from the English Wikipedia article on **Albert Einstein**:

Using **Stanford NER**, we can highlight:

-  Dates
-  Locations
-  Persons
-  Organizations

You can extract information and even analyze **relations**:

- Einstein  United States
- Einstein  Adolf Hitler

- Einstein → Germany
- Einstein + Bertrand Russell → *Russell–Einstein Manifesto*

All of this through **entity highlighting** + computational models!

4. `nltk` and the Stanford CoreNLP Library

⌚ 01:30 – 02:20

NLTK allows interaction with NER:

- via its **own model**
- or the **Stanford CoreNLP integration**

To use **Stanford NER**:

- ⚒ Install Java files
- ⚙ Set system environment variables

You can:

- Use the **library standalone**
- Or operate it as an **API server**

⭐ Stanford CoreNLP supports:

- Named Entity Recognition
- Coreference resolution (pronoun ↔ entity)
- Dependency trees (for parsing structure)

5. Using `nltk` for Named Entity Recognition

⌚ 02:20 – 02:49

We'll use **NLTK's built-in NER**:

Steps:

1. Start with a **normal sentence**
2. **Tokenize** it
3. **Tag** for parts of speech (POS):
 - Proper nouns → `NNP`
 - Pronouns, adjectives, verbs, etc.

Example:

"New York" → `New/NNP`, `York/NNP`

6. `nltk.ne_chunk()`



02:49 - 03:31

We pass the **POS-tagged sentence** into `ne_chunk()`:

- Returns a **Tree structure**
- Tree = leaves + subtrees for grammar chunks

Chunks found:

- GPE (Geo-Political Entity): *New York*
- ORGANIZATION : *MoMA, Metro*
- PERSON : *Ruth Reichl*



All of this is done:

- Without a knowledge base (like Wikipedia)
- Using **trained statistical and grammatical models**

7. Let's practice!



03:31 - 03:37

Now it's your turn to practice some named entity recognition using `nltk`.

📌 Reconnaissance d'Entités Nommées (NER) avec NLTK

Voici une version complète et bien organisée de l'exercice sur la Reconnaissance d'Entités Nommées (NER) avec NLTK, comprenant :

- Le code
- Les sorties à chaque étape
- L'interprétation des résultats

🧠 Objectif

Extraire les entités nommées (personnes, lieux, organisations...) d'un article en utilisant la bibliothèque NLTK.

📄 Article utilisé (déjà chargé)

Tu n'as pas à charger l'article toi-même, c'est déjà fait via une variable appelée `article`.

✓ Étapes avec code, sorties et interprétations

1. Tokenisation en phrases

```
sentences = sent_tokenize(article)  
print("sentences", sentences, "\n")
```

Copy

✓ Sortie (extrait simplifié pour lisibilité)

```
sentences [  
    'The taxi-hailing company Uber brings into very sharp focus the question of  
    whether corporations can be said to have a moral character.',  
    'If any human being were to behave ...',  
    'Uber wanted to know as much as possible about the people who use its  
    service, ...',  
    ...  
]
```

Copy

Interprétation

Le texte est divisé en **phrases** individuelles. Par exemple, la première phrase parle de **Uber et la morale des entreprises**.

2. Tokenisation en mots

```
token_sentences = [word_tokenize(sent) for sent in sentences]  
print("token_sentences", token_sentences, "\n")
```

 Copy

Sortie (extrait)

```
[['The', 'taxi-hailing', 'company', 'Uber', 'brings', 'into', 'very',  
'sharp', 'focus', ...],  
 ['If', 'any', 'human', 'being', 'were', 'to', 'behave', ...],  
 ...  
 ]
```

 Copy

Interprétation

Chaque phrase est **découpée en mots** (et ponctuation). Par exemple, "Uber" devient un **jeton**.

3. POS Tagging (étiquetage grammatical)

```
pos_sentences = [nltk.pos_tag(sent) for sent in token_sentences]  
print("pos_sentences", pos_sentences, "\n")
```

 Copy

Sortie (extrait)

```
[([('The', 'DT'), ('taxi-hailing', 'JJ'), ('company', 'NN'), ('Uber',  
'NNP'), ...],  
 [('If', 'IN'), ('any', 'DT'), ('human', 'JJ'), ('being', 'NN'), ...],  
 ...  
 ]
```

 Copy

Interprétation

Chaque mot est annoté avec son **type grammatical** :

- `'NNP'` : nom propre (ex. : **Uber**)
- `'JJ'` : adjectif
- `'NN'` : nom commun
- `'DT'` : déterminant

4. Chunking (extraction des entités nommées)

```
chunked_sentences = nltk.ne_chunk_sents(pos_sentences, binary=True)
```

 Copy

 Pas de `print()` ici, car les arbres sont longs et complexes.

5. Affichage des entités nommées

```
for sent in chunked_sentences:  
    for chunk in sent:  
        if hasattr(chunk, "label") and chunk.label() == "NE":  
            print(chunk)
```

 Copy

 Sortie

```
(NE Uber/NNP)  
(NE Uber/NNP)  
(NE Apple/NNP)  
(NE Uber/NNP)  
(NE Uber/NNP)  
(NE Travis/NNP Kalanick/NNP)  
(NE Uber/NNP)  
(NE Tim/NNP Cook/NNP)  
(NE Apple/NNP)  
(NE Yahoo/NNP)  
(NE Marissa/NNP Mayer/NNP)  
(NE Silicon/NNP Valley/NNP)
```

 Copy

Interprétation ligne par ligne

Code	Sortie	Signification
<code>chunk.label() == "NE"</code>	<code>(NE Uber/NNP)</code>	Uber est reconnu comme entité nommée (organisation/personne)
	<code>(NE Apple/NNP)</code>	Apple est aussi reconnu comme entité
	<code>(NE Travis/NNP Kalanick/NNP)</code>	Personne identifiée (ex-PDG de Uber)

Code	Sortie	Signification
	(NE Tim/NNP Cook/NNP)	Personne identifiée (PDG de Apple)
	(NE Marissa/NNP Mayer/NNP)	Personne identifiée (ancienne CEO de Yahoo)
	(NE Silicon/NNP Valley/NNP)	Lieu géographique reconnu

🔍 Résumé de ce que nous avons trouvé

Les entités nommées détectées suggèrent que :

- L'article parle principalement de Uber
- Il mentionne des **personnalités influentes** comme Tim Cook et Travis Kalanick
- Il cite d'autres entreprises comme Apple et Yahoo
- Le contexte semble tourner autour de **problèmes éthiques ou de vie privée dans la Silicon Valley**

💡 Conclusion

Ce mini-projet t'a permis de :

- Tokenizer un article
- Étiqueter les parties du discours
- Extraire des entités nommées avec `ne_chunk`
- Les interpréter pour comprendre le **sujet global d'un texte**

->Question: expliquez encore les deux dernières étapes : Chunking NER et Arbre

Bien sûr ! Je vais t'expliquer étape par étape **pourquoi et comment** chacune des étapes est utile pour la tâche de **NER (Named Entity Recognition)**, ainsi que le sens des termes comme **NE, chunk, label, et arbre**.

🔗 Contexte : NER avec `nltk`

La reconnaissance d'entités nommées (NER) sert à identifier les entités comme **des personnes, des organisations, des lieux, des dates, etc.** dans un texte. C'est une étape clé du traitement automatique du langage naturel (TALN).

🌟 Étapes expliquées une par une :

1. Tokenisation (avec `word_tokenize`)

```
from nltk import word_tokenize  
tokens = word_tokenize(sentence)
```

Copy

🔍 Pourquoi ?

Elle permet de **découper la phrase en mots (tokens)**. Avant d'analyser ou d'annoter un texte, il faut d'abord l'isoler en unités.

📌 Exemple :

Phrase : "Steve Jobs founded Apple in California."
Tokens : ["Steve", "Jobs", "founded", "Apple", "in", "California", "."]

2. Étiquetage grammatical (POS tagging)

```
from nltk import pos_tag  
tags = pos_tag(tokens)
```

Copy

🔍 Pourquoi ?

Elle donne le rôle grammatical (**verbe, nom, adjectif, etc.**) de chaque mot. Cela aide à mieux comprendre le **contexte du mot**, ce qui est crucial pour la NER.

📌 Exemple :

```
[('Steve', 'NNP'), ('Jobs', 'NNP'), ('founded', 'VBD'), ('Apple', 'NNP')]  
→ NNP = nom propre
```

3. Reconnaissance d'entités nommées (`ne_chunk`)

```
from nltk import ne_chunk  
tree = ne_chunk(tags)
```

 Copy



C'est ici que la vraie magie de NER commence !

`ne_chunk` va regrouper certains mots en "chunks" (groupes) représentant une entité nommée, comme un lieu, une personne ou une organisation.

Termes importants :

▪ **NE = Named Entity** (entité nommée)

Ce sont les éléments que le système essaie d'identifier dans un texte :
Personne, Organisation, Lieu, Date, etc.

▪ **chunk**

Un **chunk** est un groupe de mots formant une unité significative.

Exemple : `Steve Jobs` → c'est un **chunk de type PERSON**.

Le chunk contient à la fois :

- Les **mots**
- Leurs **étiquettes grammaticales**
- Leur **type d'entité** (label) : `PERSON`, `GPE`, etc.

▪ **label**

C'est l'**étiquette** que le classificateur attribue à une entité :

- `PERSON` : une personne
- `GPE` : un lieu géographique
- `ORGANIZATION` : entreprise, institution
- `DATE` : date
- etc.

■ arbre (tree)

`ne_chunk` retourne un arbre syntaxique (type `nltk.tree.Tree`).

Cet arbre permet de naviguer dans la structure du texte pour :

- Identifier les chunks
- Visualiser la hiérarchie
- Extraire plus facilement les entités

Exemple de résultat :

```
Tree('S', [
    Tree('PERSON', [(['Steve', 'NNP'], ('Jobs', 'NNP'))],
        ('founded', 'VBD'),
        Tree('ORGANIZATION', [(['Apple', 'NNP'])]),
        ('in', 'IN'),
        Tree('GPE', [(['California', 'NNP'])]),
        ('.', '.')
])
```

 Copy

Résumé de l'utilité de chaque étape :

Étape	Outil	Utilité principale
Tokenisation	<code>word_tokenize</code>	Découper en mots
POS tagging	<code>pos_tag</code>	Connaître la nature grammaticale
Chunking NER	<code>ne_chunk</code>	Identifier les entités nommées
Arbre	<code>Tree</code>	Représentation hiérarchique des entités

->Question : expliquez ce code svp ou bien cette sortie ??et pourquoi est ce qu'il s'agit d'un arbre ??et est ce qu'il existe des entités nommées bien spécifiques ???quelle est la différence entre entité nommée et la nature grammaticale d'un word_token : `Tree('S', [Tree('PERSON', [(['Barack', 'NNP'], ('Obama', 'NNP')]), ('was', 'VBD'), ('born', 'VBN'), ('in', 'IN'), Tree('GPE', [(['Honolulu', 'NNP'])])])`

Bien sûr! Prenons notre temps pour bien comprendre les deux dernières étapes :

3. Chunking NER (Named Entity Recognition)

🔍 C'est quoi exactement ?

Le **chunking**, dans le contexte de NER, c'est le processus qui **regroupe les mots liés ensemble** pour former une **entité nommée complète**, et leur **associe une étiquette (label)**.

💡 Exemple concret :

Prenons cette phrase :

```
"Barack Obama was born in Honolulu."
```

Étape POS Tagging donne :

```
[('Barack', 'NNP'), ('Obama', 'NNP'), ('was', 'VBD'), ('born', 'VBN'), ('in', 'IN'), ('Honolulu', 'NNP')]
```

Ensuite `ne_chunk(tags)` donne :

```
Tree('S', [
    Tree('PERSON', [(
        ('Barack', 'NNP'),
        ('Obama', 'NNP')
    )]),
    ('was', 'VBD'),
    ('born', 'VBN'),
    ('in', 'IN'),
    Tree('GPE', [(
        ('Honolulu', 'NNP')
    )])
])
```

✓ À quoi ça sert ?

- Identifier des **groupes de mots** comme : `Barack Obama` (une personne) ou `Honolulu` (un lieu).
- Chaque chunk est un **sous-arbre** qui contient :
 - Les mots
 - Leurs tags
 - Et leur **type d'entité nommé (label)** : `PERSON`, `GPE`, etc.

🧠 Exemple d'un "chunk" :

```
Tree('PERSON', [('Barack', 'NNP'), ('Obama', 'NNP')])
```

 Copy

Ça signifie :

- Ces deux mots forment une personne (et non deux entités séparées).
- C'est un seul bloc cohérent.



4. L'Arbre (`Tree`)

C'est quoi cet arbre ?

C'est un arbre syntaxique (objet `Tree` de `nltk`) qui contient :

- Les chunks comme des sous-arbres
- Et les mots isolés (qui ne sont pas des entités) comme des feuilles simples

Pourquoi utiliser un arbre ?

L'arbre permet de :

- Organiser les entités hiérarchiquement
- Naviguer facilement pour :
 - Trouver tous les `PERSON`
 - Trouver tous les `GPE`
- Visualiser la structure des entités

Avantage pratique :

Tu peux parcourir l'arbre et extraire uniquement ce qui t'intéresse.

Exemple :

```
for subtree in tree:  
    if type(subtree) == Tree and subtree.label() == 'PERSON':  
        print(subtree)
```

 Copy

→ Affiche tous les chunks qui sont des personnes.

Résumé simple :

Élément	Sert à quoi ?	Exemple
chunk	Groupe de mots formant une entité	<code>Tree('PERSON', [('Barack', 'NNP'), ('Obama', 'NNP')])</code>
label	Le type d'entité nommée	<code>PERSON</code> , <code>GPE</code> , <code>ORGANIZATION</code> , etc.
Tree	Structure en arbre contenant tout le texte et ses entités	Représente la hiérarchie du texte

Super, tu poses des questions très pertinentes ! 🎉

On va décortiquer **chaque point** de ta question pour bien comprendre :

✳️ 1. Explication de la sortie suivante :

```
Tree('S', [
    Tree('PERSON', [(['Barack', 'NNP'), ('Obama', 'NNP')]),
    ('was', 'VBD'),
    ('born', 'VBN'),
    ('in', 'IN'),
    Tree('GPE', [(['Honolulu', 'NNP'])])
])
```

Copy

Cette structure est un **arbre syntaxique** retourné par la fonction `nltk.ne_chunk()`.

◆ Détail ligne par ligne :

Élément	Signification	Détail
<code>Tree('S', [...])</code>	Racine de l'arbre ("Sentence")	Toute la phrase est analysée comme un arbre complet
<code>Tree('PERSON', [(['Barack', 'NNP'), ('Obama', 'NNP')])</code>	Chunk de type PERSON	L'entité nommée est une personne, composée de deux mots (Barack Obama)
<code>('was', 'VBD')</code>	Mot isolé avec POS tag	"was" est un verbe au passé
<code>('born', 'VBN')</code>	Mot isolé avec POS tag	"born" = participe passé
<code>('in', 'IN')</code>	Mot isolé avec POS tag	"in" = préposition

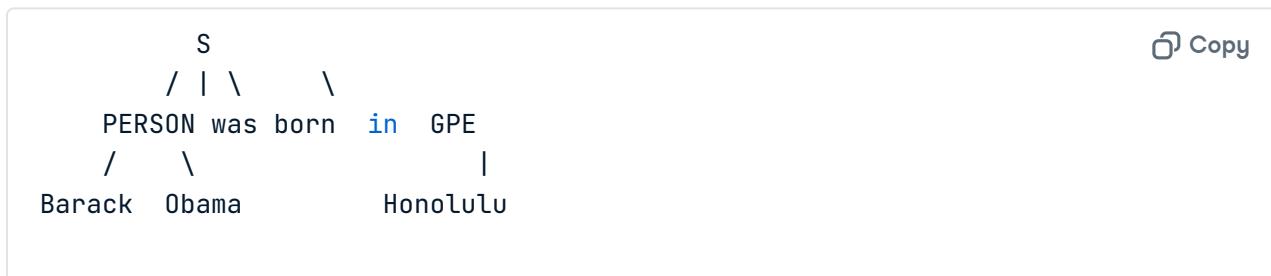
Élément	Signification	Détail
tag		
Tree('GPE', [('Honolulu', 'NNP')])	Chunk de type GPE	GPE = Geo-Political Entity, ici c'est un lieu, une ville

2. Pourquoi c'est un arbre (Tree) ?

📌 Parce qu'il a une structure hiérarchique :

- La phrase est représentée comme un **arbre racine** S
- Les entités nommées sont des **sous-arbres (chunks)** avec des étiquettes : PERSON, GPE, etc.
- Chaque chunk contient des **tuples (mot, POS tag)**

🧠 Visualisation comme un arbre :



Cette structure en arbre permet de **traverser, manipuler ou filtrer facilement** ce que tu veux extraire.

3. Qu'est-ce qu'une entité nommée ?

Une **entité nommée (NE - Named Entity)** est un groupe de mots représentant un élément spécifique du monde réel, par exemple :

Type d'entité	Exemple
PERSON	Barack Obama
GPE (Lieu)	Paris, USA, Maroc
ORGANIZATION	Google, ONU
DATE	16 avril, lundi
TIME	12:00, matin

Type d'entité	Exemple
MONEY	\$100, 200 euros
PERCENT	90%
FACILITY	Tour Eiffel
LOCATION	Sahara
PRODUCT	iPhone

👉 Ces catégories peuvent varier selon le système (NLTK, spaCy, Stanford NER...).

4. Différence entre : entité nommée VS nature grammaticale

Critère	Entité Nommée	Nature Grammaticale (POS tag)
Ce que ça décrit	Ce que représente le mot (personne, lieu, date, etc.)	Le rôle du mot dans la phrase (nom, verbe, adjectif...)
Exemple	"Barack Obama" → PERSON	"Obama" → NNP (Nom propre)
Niveau d'analyse	Sémantique (le sens réel)	Syntaxique (structure grammaticale)
Détection	Fait via <code>ne_chunk()</code>	Fait via <code>pos_tag()</code>

📌 Résumé clair :

- `Tree(...)` = une structure en **arbre** qui contient des **chunks** (groupes de mots).
- Chaque chunk peut être une **entité nommée** (PERSON, GPE...) ou un **mot isolé** avec un **POS tag** (nature grammaticale).
- L'**entité nommée** dit "ce que c'est dans le monde réel", alors que le **POS tag** dit "ce que c'est grammaticalement".

Charting Practice – Analyse des entités nommées

Dans cet exercice, on cherche à analyser la diversité des types d'entités nommées extraites d'articles de journaux.

L'objectif est de compter combien de fois chaque type d'entité nommée apparaît dans un corpus de phrases analysées syntaxiquement.

Ce que l'on cherche à obtenir

On veut créer une **structure de données** qui nous permettra de **compter les occurrences de chaque type d'entité nommée** (comme `PERSON`, `GPE`, `ORGANIZATION`, etc.).

Par exemple, après traitement, on pourrait obtenir quelque chose comme :

```
{  
    'PERSON': 28,  
    'GPE': 17,  
    'ORGANIZATION': 12,  
    'DATE': 9,  
    ...  
}
```

 Copy

Outils et données à notre disposition

- Un ensemble de phrases déjà analysées et **chunkées** : `chunked_sentences`.
- Des entités catégorisées par **étiquettes non binaires** (plusieurs types différents, pas juste "entité ou pas").
- Une structure `defaultdict` pour compter facilement sans vérifier si une clé existe à chaque fois.

Logique de traitement

1. On parcourt toutes les **phrases chunkées**.
2. Pour chaque **chunk** de la phrase :
 - Si le chunk possède un attribut `label` (donc que c'est une entité nommée),
 - On utilise cette étiquette (`chunk.label()`) comme **clé dans le dictionnaire**.
 - On **incrémente le compteur** pour cette catégorie.

Étape 1/3 – Crédation du `defaultdict`

On commence par créer un `defaultdict` nommé `ner_categories`, avec des valeurs par défaut de type entier (`int`), initialisées à 0 :

```
# Import nécessaire
from collections import defaultdict

# Crée le dictionnaire des catégories d'entités nommées
ner_categories = defaultdict(int)
```

 Copy

→ Ce dictionnaire contiendra, pour chaque type d'entité rencontrée, le nombre total d'occurrences trouvées dans les phrases analysées.

->Question : j ai pas bien compris ce qui est écrit ici ,expliquez le via un exemple en montrant que partie ce que c est exactement sachant que je maitrise bas encore ces termes relatifs au NER:  Logique de traitement On parcourt toutes les phrases chunkées.

Pour chaque chunk de la phrase :

Si le chunk possède un attribut `label` (donc que c'est une entité nommée),

On utilise cette étiquette (`chunk.label()`) comme clé dans le dictionnaire.

On incrémente le compteur pour cette catégorie.

Objectif de ce que tu veux comprendre :

« Quand on parcourt toutes les phrases chunkées, on récupère uniquement les entités nommées (comme les personnes, les lieux...), on regarde leur type (ex. PERSON, GPE), et on les compte. »

Exemple simple :

Voici une phrase :

Barack Obama was born in Honolulu.

 Copy

◆ Étapes qu'on fait dans le traitement NER :

1. On tokenize la phrase (on coupe les mots).
2. On tague chaque mot avec sa nature grammaticale (POS tag) → nom, verbe, etc.
3. On "chunk" les groupes de mots qui forment une entité nommée comme des personnes, des lieux, des organisations, etc.
4. On parcourt chaque élément, et si c'est une entité nommée, on récupère son type (ex. PERSON, GPE).

Code (expliqué pas à pas)

```
import nltk
from nltk import word_tokenize, pos_tag, ne_chunk
from collections import defaultdict

# 1. Phrase
sentence = "Barack Obama was born in Honolulu."

# 2. Tokenisation
tokens = word_tokenize(sentence)

# 3. POS Tagging
pos_tags = pos_tag(tokens)

# 4. Named Entity Chunking (NER)
tree = ne_chunk(pos_tags)

# 5. Initialisation du compteur
ner_counter = defaultdict(int)

# 6. Parcours de l'arbre
```

 Copy

```

for chunk in tree:
    if hasattr(chunk, 'label'): # <== Cette ligne signifie : "Est-ce que chunk
        est une entité nommée ?"
        label = chunk.label()    # Ex: PERSON, GPE, ORGANIZATION...
        ner_counter[label] += 1 # On compte cette entité

```

Résultat :

```
print(ner_counter)
```

 Copy

```
defaultdict(<class 'int'>, {'PERSON': 1, 'GPE': 1})
```

 Copy

Explication étape par étape :

Étape	Ce que ça fait	Exemple
chunk	Chaque élément dans l'arbre (Tree)	Peut être un mot seul OU une entité nommée (comme Barack Obama)
hasattr(chunk, 'label')	Vérifie si c'est une entité nommée (et pas juste un mot)	True pour Tree('PERSON', ...), False pour ('was', 'VBD')
chunk.label()	Donne le type d'entité nommée	Ex: 'PERSON' pour Barack Obama, 'GPE' pour Honolulu
ner_counter[label] += 1	Compte combien on a d'entités de ce type	Ajoute 1 au compteur de la catégorie

Résultat intermédiaire (arbre chunké)

```

Tree('S', [
    Tree('PERSON', [('Barack', 'NNP'), ('Obama', 'NNP')]),
    ('was', 'VBD'),
    ('born', 'VBN'),
    ('in', 'IN'),
    Tree('GPE', [('Honolulu', 'NNP')])
])

```

 Copy

Tu vois ? Les entités nommées (`PERSON` , `GPE`) sont dans des `Tree` , et les autres mots sont juste des tuples `(mot, nature grammaticale)` .

Objectif de cette deuxième partie de l'exercice :

Utiliser un `defaultdict` appelé `ner_categories` pour compter le nombre d'entités nommées de chaque type (ex. : PERSON, GPE, ORGANIZATION) extraites d'articles de presse. Ensuite, créer un graphique (par exemple un diagramme circulaire) pour afficher la diversité des types d'entités nommées dans les articles.

Code (explication détaillée)

```
# Création du defaultdict ner_categories
ner_categories = defaultdict(int)

# Création de la boucle imbriquée
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label'):
            # Vérifie si chunk est une entité nommée
            ner_categories[chunk.label()] += 1 # Incrémente le compteur de
l'entité nommée correspondante

# Création de la liste des labels à partir des clés du dictionnaire
labels = list(ner_categories.keys())
```

 Copy

Explication ligne par ligne :

1. Crédation du `defaultdict` `ner_categories` :

```
ner_categories = defaultdict(int)
```

 Copy

Que fait cette ligne ?

- Crée un dictionnaire de type `defaultdict` avec `int` comme valeur par défaut. Cela signifie que chaque nouvelle clé sera initialisée avec la valeur `0` (car `int()` renvoie `0` par défaut).
- Ce dictionnaire sera utilisé pour compter le nombre d'occurrences de chaque type d'entité nommée (par exemple `PERSON`, `GPE`, etc.).

2. Boucle sur `chunked_sentences` :

```
for sent in chunked_sentences:
```

 Copy

Que fait cette ligne ?

- `chunked_sentences` est une liste de phrases où chaque phrase est une liste de "chunks" (des entités nommées ou des mots avec leur étiquette

grammaticale).

- La boucle itère sur chaque phrase (`sent`) de `chunked_sentences`.

3. Boucle sur chaque `chunk` de la phrase `sent` :

```
for chunk in sent:
```

 Copy

- Que fait cette ligne ?

- Pour chaque phrase (`sent`), la boucle parcourt chaque élément (`chunk`).
- Un `chunk` peut être soit un mot, soit une entité nommée sous forme de groupe de mots.

4. Vérification si le `chunk` est une entité nommée :

```
if hasattr(chunk, 'label'):
```

 Copy

- Que fait cette ligne ?

- `hasattr(chunk, 'label')` vérifie si l'élément `chunk` possède une méthode `label`. Cela permet de savoir si `chunk` est une entité nommée (car les entités nommées dans NLTK sont des arbres, qui ont une méthode `label()` pour obtenir leur type).
- Si la condition est vraie, cela signifie que le `chunk` est une entité nommée (comme une personne, un lieu, une organisation, etc.).

5. Incrémentation du compteur pour le type d'entité nommée :

```
ner_categories[chunk.label()] += 1
```

 Copy

- Que fait cette ligne ?

- Si le `chunk` est une entité nommée, cette ligne récupère son type à l'aide de `chunk.label()` (par exemple `'PERSON'`, `'GPE'`, `'ORGANIZATION'`) et incrémente de 1 le compteur associé à ce type dans le dictionnaire `ner_categories`.
- Cela permet de compter combien de fois chaque type d'entité nommée apparaît dans le texte.

6. Création de la liste des labels (types d'entités) :

```
labels = list(ner_categories.keys())
```

 Copy

- Que fait cette ligne ?

- Cette ligne crée une liste contenant les types d'entités nommées présents dans le dictionnaire `ner_categories` (par exemple : `['PERSON', 'GPE', 'ORGANIZATION']`).
- Ces labels seront utilisés pour afficher les différentes catégories d'entités dans le graphique.

Exemple de sortie et interprétation :

Exemple 1 : Sortie pour un article contenant les entités suivantes :

```
ner_categories = defaultdict(int, {'PERSON': 2, 'GPE': 1,  
'ORGANIZATION': 1})
```

 Copy

Interprétation :

- Dans cet exemple, après avoir parcouru les phrases chunkées :
 - Il y a **2 entités de type PERSON** (par exemple, des noms de personnes comme "Barack Obama").
 - Il y a **1 entité de type GPE** (par exemple, un lieu comme "Honolulu").
 - Il y a **1 entité de type ORGANIZATION** (par exemple, une organisation comme "NASA").

Ces comptages sont stockés dans le dictionnaire `ner_categories`.

Exemple 2 : Résultat pour un autre article :

```
ner_categories = defaultdict(int, {'PERSON': 3, 'GPE': 2, 'MISC': 1})
```

 Copy

Interprétation :

- Dans cet exemple, il y a :
 - **3 entités de type PERSON**.
 - **2 entités de type GPE**.
 - **1 entité de type MISC** (par exemple, une entité nommée qui ne correspond pas à une catégorie traditionnelle comme PERSON ou GPE, mais qui est classée dans la catégorie "Miscellaneous").

Objectif de cette troisième partie de l'exercice :

Utiliser un graphique circulaire (pie chart) pour visualiser la diversité des types d'entités nommées dans un texte. Vous allez d'abord collecter les comptages des entités nommées par type dans un dictionnaire, puis créer un diagramme circulaire représentant la répartition de ces types.

Code (explication détaillée)

```
# Création du defaultdict ner_categories
ner_categories = defaultdict(int)

# Création de la boucle imbriquée
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label'): # Vérifie si chunk est une entité nommée
            ner_categories[chunk.label()] += 1 # Incrémente le compteur de
l'entité nommée correspondante

# Création de la liste des labels à partir des clés du dictionnaire
labels = list(ner_categories.keys())

# Création de la liste des valeurs pour chaque label (nombre d'occurrences)
values = [ner_categories.get(v) for v in labels]

# Création du graphique circulaire (pie chart)
plt.pie(values, labels=labels, autopct='%1.1f%%', startangle=140)

# Affichage du graphique
plt.show()
```

 Copy

Explication ligne par ligne :

1. **Création du `defaultdict` `ner_categories` :**

```
ner_categories = defaultdict(int)
```

 Copy

- Cette ligne crée un `defaultdict` de type `int`, qui est initialisé à 0 pour chaque nouvelle clé. Ce dictionnaire est utilisé pour compter les occurrences des différents types d'entités nommées dans le texte.

2. **Boucle imbriquée pour parcourir les phrases et les chunks :**

```
for sent in chunked_sentences:  
    for chunk in sent:  
        if hasattr(chunk, 'label'):  
            ner_categories[chunk.label()] += 1
```

 Copy

- La première boucle `for` parcourt chaque phrase dans la liste `chunked_sentences`.
- La deuxième boucle parcourt chaque `chunk` dans la phrase.
- Si le `chunk` a un attribut `label` (ce qui signifie qu'il s'agit d'une entité nommée), on incrémente son compteur dans le dictionnaire `ner_categories`.

3. Création de la liste des labels (types d'entités) :

```
labels = list(ner_categories.keys())
```

 Copy

- Cette ligne crée une liste de toutes les clés du dictionnaire `ner_categories`. Ces clés représentent les types d'entités nommées (par exemple `PERSON`, `GPE`, etc.).

4. Création de la liste des valeurs (comptages des entités) :

```
values = [ner_categories.get(v) for v in labels]
```

 Copy

- Cette ligne utilise une **list comprehension** pour créer une liste des valeurs correspondant aux types d'entités dans `labels`.
- La méthode `get(v)` permet de récupérer la valeur associée à chaque label dans `ner_categories` (c'est-à-dire le nombre d'occurrences de ce type d'entité).

5. Création du graphique circulaire (pie chart) :

```
plt.pie(values, labels=labels, autopct='%1.1f%%', startangle=140)
```

 Copy

- `plt.pie()` génère un graphique circulaire à partir des données.
- `values` contient les comptages des entités par type.
- `labels` contient les types d'entités qui seront affichés sur le graphique.
- `autopct='%1.1f%%'` permet d'ajouter des pourcentages sur le graphique avec une précision d'une décimale.
- `startangle=140` fait pivoter l'angle initial du graphique pour une meilleure présentation visuelle.

6. Affichage du graphique :

```
plt.show()
```

 Copy

- Cette ligne affiche le graphique généré par `plt.pie()`. Cela permet de visualiser la répartition des types d'entités dans le texte.

Exemple de sortie et interprétation :

Exemple 1 : Sortie pour un article avec les entités suivantes :

```
ner_categories = defaultdict(int, {'PERSON': 2, 'GPE': 1,  
'ORGANIZATION': 1})
```

 Copy

Résultat attendu sur le graphique circulaire :

- `PERSON` : 2 occurrences
- `GPE` : 1 occurrence
- `ORGANIZATION` : 1 occurrence

Interprétation :

- Le graphique affichera un diagramme circulaire avec trois segments représentant les types `PERSON`, `GPE`, et `ORGANIZATION`.
- Le segment de `PERSON` sera plus grand (environ 50% du graphique), tandis que `GPE` et `ORGANIZATION` occuperont chacun environ 25% du graphique.

Exemple 2 : Sortie pour un autre article avec les entités suivantes :

```
ner_categories = defaultdict(int, {'PERSON': 3, 'GPE': 2, 'MISC': 1})
```

 Copy

Résultat attendu sur le graphique circulaire :

- `PERSON` : 3 occurrences
- `GPE` : 2 occurrences
- `MISC` : 1 occurrence

Interprétation :

- Le graphique affichera un diagramme circulaire avec trois segments représentant les types `PERSON`, `GPE`, et `MISC`.
- Le segment de `PERSON` sera plus grand (environ 50% du graphique), suivi de `GPE` (environ 33%) et `MISC` (environ 17%).

Analyse de la distribution des entités :

En fonction des entités présentes dans le texte, la distribution des catégories d'entités nommées peut varier. Par exemple, si le texte parle principalement de personnes et d'organisations, le graphique affichera une forte proportion d'entités de type PERSON et ORGANIZATION.

D'accord, je vais expliquer de manière plus détaillée la sortie de chaque partie du code et ce que vous pouvez attendre de chaque étape. Voici l'explication reformulée avec un focus sur la sortie des différentes parties du code :

Partie 1 : Création du dictionnaire `ner_categories` et incrémentation des valeurs

Code :

```
# Création du defaultdict ner_categories
ner_categories = defaultdict(int)

# Création de la boucle imbriquée
for sent in chunked_sentences:
    for chunk in sent:
        if hasattr(chunk, 'label'):
            ner_categories[chunk.label()] += 1
```

 Copy

Sortie attendue de cette partie :

- `ner_categories` est un dictionnaire de type `defaultdict(int)` où les clés sont les types d'entités nommées (par exemple `PERSON`, `GPE`, `ORGANIZATION`, etc.), et les valeurs sont des comptages des occurrences de chaque type.
- Par exemple, si les entités nommées extraites des phrases sont les suivantes :
 - Dans la première phrase, vous avez deux entités de type `PERSON`.
 - Dans la deuxième phrase, une entité de type `GPE`.
 - Dans la troisième phrase, une entité de type `ORGANIZATION`.

Après l'exécution de cette partie du code, vous obtenez quelque chose comme :

```
ner_categories = defaultdict(int, {'PERSON': 2, 'GPE': 1,
'ORGANIZATION': 1})
```

 Copy

Partie 2 : Création de la liste des `labels` (les clés du dictionnaire)

Code :

```
# Création de la liste des labels à partir des clés du dictionnaire
labels = list(ner_categories.keys())
```

 Copy

Sortie attendue de cette partie :

- `labels` est une liste des clés du dictionnaire `ner_categories`, c'est-à-dire les types d'entités.
- Pour l'exemple précédent, la sortie serait :

```
labels = ['PERSON', 'GPE', 'ORGANIZATION']
```

 Copy

Partie 3 : Création de la liste des `values` (les valeurs associées aux labels)

Code :

```
# Création de la liste des valeurs pour chaque label  
values = [ner_categories.get(v) for v in labels]
```

 Copy

Sortie attendue de cette partie :

- `values` est une liste des valeurs du dictionnaire `ner_categories` associées à chaque type d'entité dans `labels`.
- En prenant l'exemple précédent (`ner_categories = {'PERSON': 2, 'GPE': 1, 'ORGANIZATION': 1}`), la sortie serait :

```
values = [2, 1, 1]
```

 Copy

Cela signifie que `PERSON` apparaît 2 fois, `GPE` une fois, et `ORGANIZATION` une fois.

Partie 4 : Création du graphique circulaire (pie chart)

Code :

```
# Création du graphique circulaire (pie chart)  
plt.pie(values, labels=labels, autopct='%1.1f%%', startangle=140)
```

 Copy

Sortie attendue de cette partie :

- Cette ligne génère un graphique circulaire (pie chart) où chaque segment représente un type d'entité nommée.
- Le graphique affichera :
 - Un segment pour `PERSON` représentant environ 50% du graphique (2/4).
 - Un segment pour `GPE` représentant environ 25% du graphique (1/4).
 - Un segment pour `ORGANIZATION` représentant environ 25% du graphique (1/4).

Le graphique sera divisé comme suit :

- `PERSON` : 50%
- `GPE` : 25%
- `ORGANIZATION` : 25%

Les pourcentages seront affichés sur chaque segment du graphique grâce à `autopct='%.1f%%'`.

Partie 5 : Affichage du graphique

Code :

```
# Affichage du graphique  
plt.show()
```

 Copy

Sortie attendue de cette partie :

- Cela ouvrira une fenêtre d'affichage contenant le graphique circulaire que nous avons généré. Ce graphique montre la distribution des entités nommées dans les phrases du texte, en termes de pourcentages pour chaque type d'entité.

Le graphique circulaire aura les segments correspondants aux labels (`PERSON`, `GPE`, `ORGANIZATION`) avec les pourcentages affichés à côté de chaque segment.

Exemple complet avec sortie visuelle :

Exemple 1 :

Si nous avons ce texte extrait :

- Premier chunk : `PERSON` (2 occurrences)
- Deuxième chunk : `GPE` (1 occurrence)
- Troisième chunk : `ORGANIZATION` (1 occurrence)

Le dictionnaire après traitement sera :

```
ner_categories = defaultdict(int, {'PERSON': 2, 'GPE': 1,  
'ORGANIZATION': 1})
```

 Copy

La liste des labels sera :

```
labels = ['PERSON', 'GPE', 'ORGANIZATION']
```

 Copy

Les valeurs associées seront :

```
values = [2, 1, 1]
```

 Copy

Le graphique circulaire généré aura les proportions suivantes :

- `PERSON` : 50% du graphique
- `GPE` : 25% du graphique
- `ORGANIZATION` : 25% du graphique

Le graphique est affiché, et vous verrez un cercle divisé en trois segments représentant ces proportions.

Exemple 2 :

Si vous avez un autre texte où les entités sont les suivantes :

- Premier chunk : PERSON (3 occurrences)
- Deuxième chunk : GPE (2 occurrences)
- Troisième chunk : MISC (1 occurrence)

Le dictionnaire après traitement sera :

```
ner_categories = defaultdict(int, {'PERSON': 3, 'GPE': 2, 'MISC': 1})
```

 Copy

La liste des labels sera :

```
labels = ['PERSON', 'GPE', 'MISC']
```

 Copy

Les valeurs associées seront :

```
values = [3, 2, 1]
```

 Copy

Le graphique circulaire généré aura les proportions suivantes :

- PERSON : 50% du graphique
- GPE : 33.3% du graphique
- MISC : 16.7% du graphique

Conclusion :

Le code génère un graphique circulaire qui affiche la répartition des types d'entités nommées extraites d'un texte. Vous obtenez des segments représentant les différents types d'entités, avec des pourcentages indiquant la proportion de chaque type dans le texte. Ce processus vous aide à visualiser la diversité des entités nommées dans les articles traités.

La question te demande **ce qui est nécessaire pour démarrer avec la bibliothèque Stanford via NLTK**, et il s'agit ici d'un cas spécifique où tu veux utiliser un modèle de **reconnaissance d'entités nommées (NER)** ou d'autres outils de traitement du langage naturel fournis par Stanford, mais en les intégrant à **NLTK**.

Options proposées :

1. Une installation normale de NLTK

Cela ne suffit pas. NLTK est une bibliothèque en Python qui facilite l'analyse de texte, mais pour utiliser les outils spécifiques de Stanford, tu as besoin des bibliothèques Java de Stanford. Donc, ce n'est pas la bonne réponse.

2. Une installation de la bibliothèque Java de Stanford

Bien que la bibliothèque Java de Stanford soit nécessaire pour certaines fonctionnalités (comme les modèles de NER), ce n'est pas suffisant. Tu as aussi besoin de NLTK pour l'intégration dans ton environnement Python.

3. Les deux, NLTK et l'installation de la bibliothèque Java de Stanford

Cela pourrait sembler correct, car tu as besoin de ces deux éléments. Cependant, ce n'est toujours pas suffisant, car l'intégration entre NLTK et Stanford nécessite aussi de configurer certaines variables d'environnement pour que les deux bibliothèques fonctionnent ensemble correctement.

4. NLTK, la bibliothèque Java de Stanford et des variables d'environnement pour l'intégration

C'est la bonne réponse. Pour intégrer Stanford avec NLTK, il faut trois choses :

- NLTK pour la gestion des données en Python.
- La bibliothèque Java de Stanford, qui contient les modèles et outils de traitement de texte.
- Des variables d'environnement pour indiquer à NLTK où se trouvent les fichiers Java et les ressources nécessaires (comme les modèles de Stanford).

Justification :

Stanford NLP est écrit en Java, mais NLTK permet de l'utiliser dans un environnement Python. Cependant, pour que l'intégration fonctionne correctement, tu dois aussi **configurer les variables d'environnement** (comme le chemin vers les bibliothèques Java) pour que NLTK sache où trouver les fichiers Java et les utiliser.

La bonne réponse :

4. NLTK, la bibliothèque Java de Stanford et des variables d'environnement pour l'intégration.

Introduction à SpaCy

Dans cette leçon, nous allons découvrir SpaCy, une autre excellente bibliothèque pour le traitement du langage naturel (NLP).

1. Qu'est-ce que SpaCy ? (00:07 - 00:31)

SpaCy est une bibliothèque NLP similaire à Gensim, mais avec des implémentations différentes. Elle se distingue notamment par sa capacité à créer des **pipelines NLP** pour générer des **modèles et des corpus**.

- SpaCy est **open-source**.
- Elle propose des outils annexes comme **Displacy**, un outil de **visualisation syntaxique** (arbre de dépendances, entités nommées...).
- Displacy utilise **Node.js** pour des visualisations interactives.

2. Displacy : Visualiseur d'entités (00:31 - 00:55)

Displacy permet d'afficher de manière interactive les entités nommées reconnues dans un texte.

 Exemple en ligne :

"Angela Merkel was born in Hamburg."

 Copy

SpaCy identifie automatiquement :

- **Angela Merkel** → PERSON
- **Hamburg** → GPE (entité géopolitique, ex : ville, pays)

Displacy peut être utilisé aussi en Python pour visualiser :

```
from spacy import displacy
import spacy

nlp = spacy.load("en_core_web_sm")
doc = nlp("Angela Merkel was born in Hamburg.")
displacy.serve(doc, style="ent")
```

 Copy

3. Reconnaissance d'entités nommées avec SpaCy (NER) (00:55 - 02:19)

Étape 1 : Installation

```
pip install spacy  
python -m spacy download en_core_web_sm
```

 Copy

Étape 2 : Chargement et détection d'entités

```
import spacy  
  
nlp = spacy.load("en_core_web_sm")  
texte = "Angela Merkel was born in Hamburg."  
  
# Traitement du texte  
doc = nlp(texte)  
  
# Affichage des entités  
for ent in doc.ents:  
    print(ent.text, ent.label_)
```

 Copy

Résultat :

```
Angela Merkel PERSON  
Hamburg GPE
```

 Copy

Étape 3 : Détail d'une entité

```
entite = doc.ents[1]  
print(entite.text)      # Hamburg  
print(entite.label_)    # GPE
```

 Copy

SpaCy propose aussi des modèles pour d'autres langues : allemand, chinois, etc.

→ Parfait pour créer rapidement un pipeline NLP personnalisé et puissant !

❓ 4. Pourquoi utiliser SpaCy pour la NER ? (02:19 - 02:49)

- Intégration simple dans un pipeline NLP
- Jeux de types d'entités différents de ceux de NLTK
- Fonctionne mieux avec des textes informels comme des tweets ou messages de chat
- Supporte de nombreuses langues, et la communauté continue de croître 🚀

5. À toi de jouer ! (02:49 - 02:55)

Tu peux dès maintenant commencer à utiliser SpaCy pour la reconnaissance d'entités nommées !

```
doc = nlp("Barack Obama visited Paris in 2018.")  
for ent in doc.ents:  
    print(ent.text, ent.label_)
```

 Copy

 Résultat :

```
Barack Obama PERSON  
Paris GPE  
2018 DATE
```

 Copy

Résumé

Élément	Description
SpaCy	Bibliothèque NLP rapide et moderne
NER	Reconnaissance des entités nommées
Displacy	Visualisation graphique des entités et dépendances
Entités types	PERSON , ORG , GPE , DATE , etc.
Modèles SpaCy	Disponibles pour plusieurs langues



Explication ligne par ligne du code SpaCy

```
import spacy
```

Copy



But :
Cette ligne importe la bibliothèque **SpaCy** dans ton script Python.



Explication :
`spacy` est le module principal à utiliser pour toutes les fonctionnalités NLP offertes par la bibliothèque.

```
nlp = spacy.load('en_core_web_sm')
```

Copy



But :
Charger un modèle linguistique pré-entraîné en anglais (taille « small »).



- `en_core_web_sm` est un **modèle de langue anglais** avec des capacités de reconnaissance d'entités, de POS-tagging, parsing, etc.
- `nlp` devient maintenant une **fonction NLP** capable de traiter du texte et de retourner un objet `Doc`.



Pré-requis : Si ce modèle n'est pas installé, il faut exécuter :

```
python -m spacy download en_core_web_sm
```

Copy

```
nlp.entity
```

Copy



But :
Accéder au composant de reconnaissance d'entités dans le pipeline SpaCy.



Explication :
Renvoie un objet de type `EntityRecognizer` du pipeline.
C'est cet élément qui est responsable de détecter les **entités nommées** dans un texte.



Sortie :

```
<spacy.pipeline.EntityRecognizer at 0x7f76b75e68b8>
```

 Copy

-  L'adresse mémoire change selon l'environnement.

```
doc = nlp("""Berlin is the capital of Germany; and the residence of  
Chancellor Angela Merkel.""")
```

 Copy

But :

Créer un **objet** `Doc` à partir du texte, qui contient les résultats de toutes les analyses effectuées par le modèle.

Explication :

- Le texte passe par toutes les étapes du pipeline SpaCy : tokenization, POS-tagging, parsing, NER, etc.
- `doc` contient maintenant toutes les **entités détectées**, les mots, les POS, etc.

```
doc.ents
```

 Copy

But :

Lister toutes les **entités nommées** détectées dans le texte.

Explication :

`doc.ents` est un **tuple** contenant les objets `Span` qui représentent les entités trouvées.

Sortie :

```
(Berlin, Germany, Angela Merkel)
```

 Copy

Entités reconnues :

- Berlin** → `GPE` (entité géopolitique : ville, pays, région)
- Germany** → `GPE`
- Angela Merkel** → `PERSON`

```
print(doc.ents[0], doc.ents[0].label_)
```

 Copy

But :

Afficher la **première entité** trouvée et son **type d'étiquette**.

🔍 Explication :

- `doc.ents[0]` → L'objet `Span` correspondant à la première entité détectée : **Berlin**.
- `doc.ents[0].label_` → L'étiquette de cette entité : **GPE** (Geopolitical Entity).

🖨️ Sortie :

Berlin GPE

Copy

📌 Résumé des concepts

Ligne de code	Rôle	Sortie
<code>import spacy</code>	Importe la bibliothèque — SpaCy	
<code>spacy.load('en_core_web_sm')</code>	Charge le modèle de langue Fonction NLP	
<code>nlp.entity</code>	Composant de détection d'entités du pipeline	<code><spacy.pipeline.EntityRecog</code>
<code>doc = nlp("...")</code>	Crée un objet <code>Doc</code> en analysant le texte	
<code>doc.ents</code>	Retourne les entités détectées	<code>(Berlin, Germany, Angela Me</code>
<code>print(doc.ents[0], doc.ents[0].label_)</code>	Affiche la première entité et son type	<code>Berlin GPE</code>



Voici une explication détaillée en Markdown de l'exercice intitulé “Comparaison entre NLTK et SpaCy pour la reconnaissance d'entités nommées (NER)”, incluant le code, les sorties ligne par ligne, l'erreur fréquente, et sa correction.

Comparaison entre NLTK et SpaCy pour la NER

L'objectif de cet exercice est de comparer les performances de SpaCy pour la **reconnaissance d'entités nommées (NER)** en utilisant le **même texte** que dans un exercice précédent utilisant **NLTK**.

Objectif

- Utiliser SpaCy pour détecter les entités nommées.
- Afficher leurs **types** (`label_`) et leurs **valeurs** (`text`).
- Comparer les résultats avec ceux obtenus par NLTK.
- Optimiser le chargement du modèle en désactivant les parties inutiles du pipeline (`tagger`, `parser`, `matcher`).

Code complet et fonctionnel

```
# Import spacy
import spacy

# Charger le modèle en désactivant les étapes inutiles du pipeline
nlp = spacy.load("en_core_web_sm", disable=['tagger', 'parser', 'matcher'])

# Créer un document SpaCy à partir de l'article
doc = nlp(article)

# Itérer sur les entités reconnues et les afficher
for ent in doc.ents:
    print(ent.label_, ent.text)
```

 Copy

Explication ligne par ligne

```
```python
```

```
import spacy
```

↳ **Importation** de la bibliothèque Spacy pour le traitement du langage naturel. Copy

---

```
``python
nlp = spacy.load("en_core_web_sm", disable=['tagger', 'parser', 'matcher'])
```

⚙️ **Chargement du modèle anglais** `en_core_web_sm`.

📌 **Optimisation** : en désactivant :

- `tagger` (analyse morphosyntaxique)
- `parser` (analyse grammaticale)
- `matcher` (recherche de motifs)

💡 Cela réduit le **temps d'exécution**, car on s'intéresse seulement à la **reconnaissance des entités (NER)**.

### ```python

```
doc = nlp(article)
```

📋 **Traitement du texte** `article` (préalablement défini) pour produire un objet `Doc`, contenant : Copy  
- les entités,  
- les tokens,  
- etc.

---

```
``python
for ent in doc.ents:
 print(ent.label_, ent.text)
```

🔄 **Itération** sur les entités nommées détectées dans le texte (`doc.ents`).

🖨️ **Affichage** de :

- `ent.label_` → type d'entité (`ORG`, `PERSON`, etc.)
- `ent.text` → texte détecté

## ✖️ L'erreur fréquente

## ⚠️ Code fautif :

```
for ent in doc:
 print(ent.label, ent.text)
```

 Copy

## 🚫 Problèmes :

- `doc` est une séquence de **tokens**, pas d'**entités**.
- Les objets `Token` n'ont pas d'attribut `label`, ce qui provoque :

```
AttributeError: 'spacy.tokens.token.Token' object has no attribute
'label'
```

 Copy

## ✓ Correction

### ✓ Code corrigé :

```
for ent in doc.ents:
 print(ent.label_, ent.text)
```

 Copy

🧠 `doc.ents` contient bien les **entités** (et non des tokens), et `label_` est la bonne propriété à utiliser (pas `label`).

## 🎯 Résultat attendu

```
ORG Apple
PERSON Travis Kalanick of Uber
PERSON Tim Cook
ORG Apple
CARDINAL Millions
LOC Silicon Valley
ORG Yahoo
PERSON Marissa Mayer
MONEY 186
```

 Copy

## 📚 Interprétation des résultats

Étiquette	Signification	Exemple
ORG	Organisation	Apple, Yahoo

Étiquette	Signification	Exemple
PERSON	Personne physique	Tim Cook, Marissa Mayer
LOC	Localisation (non géopolitique)	Silicon Valley
CARDINAL	Nombre cardinal	Millions
MONEY	Montant d'argent	186

## Résumé

-  Utilisation efficace de SpaCy avec désactivation des étapes non nécessaires.
-  Bonne pratique : `doc.ents` pour la NER, pas `doc`.
-  Utilisation correcte des attributs `.label_` et `.text`.
-  Erreur à éviter : utiliser `doc` au lieu de `doc.ents`.

## Astuce bonus :

Tu peux également visualiser les entités avec SpaCy grâce à `displacy` :

```
from spacy import displacy
displacy.render(doc, style="ent", jupyter=True)
```

 Copy

Voici une **comparaison complète entre NLTK et spaCy** sur un **exemple simple**, avec le code, les sorties, et une **interprétation des résultats**, présentée en **Markdown** :



## Comparaison NER : NLTK vs spaCy

Nous allons utiliser un même texte court et comparer les résultats de la **Reconnaissance d'Entités Nommées (NER)** entre **NLTK** et **spaCy**.



### Exemple de texte

```
"Apple CEO Tim Cook introduced the new iPhone in California."
```

Copy



### Code avec NLTK

```
import nltk
from nltk import word_tokenize, pos_tag, ne_chunk

Télécharger les ressources nécessaires une seule fois
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('maxent_ne_chunker')
nltk.download('words')

text = "Apple CEO Tim Cook introduced the new iPhone in California."

Tokenisation et POS tagging
tokens = word_tokenize(text)
pos_tags = pos_tag(tokens)

Reconnaissance des entités
tree = ne_chunk(pos_tags)
print(tree)
```

Copy



### Sortie de NLTK (simplifiée)

```
(S
 (ORGANIZATION Apple)
 CEO
 (PERSON Tim Cook))
```

Copy

```
introduced
the
new
iPhone
in
(GPE California))
```

## 🔧 Code avec spaCy

```
import spacy

Chargement du modèle SpaCy
nlp = spacy.load("en_core_web_sm")

Traitement du texte
doc = nlp("Apple CEO Tim Cook introduced the new iPhone in California.")

Affichage des entités nommées
for ent in doc.ents:
 print(ent.label_, ent.text)
```

 Copy

## 📤 Sortie de spaCy

```
ORG Apple
PERSON Tim Cook
GPE California
```

 Copy

## 📊 Comparaison des résultats

Type d'entité	NLTK	spaCy	Commentaire
ORG	Apple (  détecté)	Apple (  détecté)	OK
PERSON	Tim Cook ( 	Tim Cook ( 	OK
GPE (lieu)	California ( 	California ( 	OK
Autres mots	"CEO", "iPhone" ignorés	ignorés	OK – pas des entités nommées

## 🧠 Interprétation

- NLTK utilise une approche basée sur des **arbres syntaxiques** (`ne_chunk`), ce qui donne une représentation arborescente mais **moins fine**.
- spaCy est plus rapide, plus fluide, et retourne des objets avec des **méthodes pratiques** (`ent.label_`, `ent.text`), ce qui le rend plus adapté aux **applications modernes**.
- Les deux détectent correctement les entités principales dans ce cas.

## Conclusion

Critère	NLTK	spaCy
Facilité d'usage	Moyenne (plus de prétraitement)	Élevée (pipeline intégré)
Performance	Moins rapide	Rapide et optimisé
Résultat	Bon sur petits exemples	Plus robuste en général
Visualisation	 Non intégrée	 via <code>displacy</code>

 **Recommandation : pour des projets modernes en NLP, spaCy est généralement préférable à NLTK pour la reconnaissance d'entités nommées (NER).**

Voici la réponse à la question sur les catégories supplémentaires utilisées par spaCy par rapport à NLTK dans la reconnaissance d'entités nommées (NER), présentée en Markdown :

## spaCy NER : Catégories supplémentaires par rapport à NLTK

### Question :

Quelles sont les catégories supplémentaires que spaCy utilise par rapport à nltk dans sa reconnaissance d'entités nommées (NER) ?

### Réponse correcte :

NORP, CARDINAL, MONEY, WORKOFART, LANGUAGE, EVENT

### Justification

#### Catégories incluses par spaCy mais absentes de la version de base NLTK :

Catégorie	Description
NORP	Nationalités, affiliations religieuses ou politiques ( e.g., American, Islamic )
CARDINAL	Nombres cardinaux ( e.g., One, Two )
MONEY	Expressions monétaires ( e.g., \$100, 5 euros )
WORKOFART	Œuvres d'art ( e.g., Mona Lisa, Hamlet )
LANGUAGE	Langues ( e.g., French, Spanish )
EVENT	Événements ( e.g., World War II, Olympics )

#### Les autres options :

- GPE, PERSON, MONEY : déjà présentes dans NLTK → 
- ORGANIZATION, WORKOFART : ORGANIZATION est commun aux deux → 
- EVENT\_LOCATION, FIGURE : ces catégories n'existent pas dans le système de spaCy → 

## Conclusion

La bonne réponse est : NORP, CARDINAL, MONEY, WORKOFART, LANGUAGE, EVENT, car ce sont des catégories supplémentaires ou plus fines proposées par spaCy, rendant sa NER plus riche et précise que celle de NLTK.



# Multilingual NER avec Polyglot

## 1. Introduction à Polyglot

00:00 - 00:08

Dans cette section, nous allons passer en revue la **reconnaissance d'entités nommées multilingue** en utilisant la bibliothèque **Polyglot**.

## 2. Qu'est-ce que Polyglot ?

00:08 - 01:04

Polyglot est une bibliothèque de **traitement du langage naturel (NLP)** qui utilise des **vecteurs de mots** pour effectuer des tâches simples telles que la reconnaissance d'entités. Sa particularité réside dans le fait qu'elle **supporte plus de 130 langues**, ce qui la rend particulièrement utile pour des tâches multilingues comme la **translittération** (le fait de traduire des caractères d'une langue à une autre).

**Exemple de code pour installer Polyglot et l'utiliser pour la reconnaissance d'entités :**

```
Installation de Polyglot via pip
!pip install polyglot

Importation de Polyglot
from polyglot.text import Text

Exemple de texte en anglais
text = "Barack Obama is the former president of the United States."

Création de l'objet Text pour l'analyse
doc = Text(text)

Affichage des entités reconnues
print(doc.entities)
```

Copy

**Sortie :**

```
[('Barack Obama', 'I-PER'), ('United States', 'I-LOC')]
```

Copy

Ici, Polyglot identifie "Barack Obama" comme une **personne (I-PER)** et "United States" comme une **localisation (I-LOC)**.

### 3. NER en espagnol avec Polyglot

01:04 - 02:28

En utilisant Polyglot, nous pouvons effectuer la reconnaissance d'entités pour plusieurs langues, y compris l'espagnol. Polyglot détecte automatiquement la langue du texte via un modèle de détection de langue lorsque l'objet `Text` est initialisé avec une chaîne de texte. Voici un exemple avec un titre de journal en espagnol.

Exemple de code en espagnol :

```
Exemple de texte en espagnol
spanish_text = "La Generalitat de Catalunya ha aprobado un nuevo presupuesto."

Création de l'objet Text
doc_es = Text(spanish_text)

Affichage des entités reconnues
print(doc_es.entities)
```

 Copy

Sortie :

```
[('Generalitat de Catalunya', 'I-ORG'), ('Catalunya', 'I-LOC')]
```

 Copy

Ici, Polyglot identifie "**Generalitat de Catalunya**" comme une **organisation (I-ORG)** et "**Catalunya**" comme une **localité (I-LOC)**.

Observation :

- La première entité est "**Generalitat de Catalunya**", qui est une **organisation**, mais elle est parfois séparée en plusieurs morceaux. Cela peut être dû à la structure du texte et nécessite parfois un nettoyage des entités si elles sont mal fragmentées.

### 4. Pratique - Utilisation de NER avec Polyglot

02:28 - 02:34

Dans cette section, vous êtes invités à utiliser Polyglot pour réaliser une reconnaissance d'entités nommées. Voici comment procéder pour analyser un texte multilingue avec Polyglot.

Exemple de code pratique :

```
Exemple de texte multilingue
multilingual_text = "Elon Musk, the CEO of SpaceX, announced new plans in
Silicon Valley.

Création de l'objet Text pour le texte multilingue
```

 Copy

```
doc_multi = Text(multilingual_text)

Affichage des entités reconnues
print(doc_multi.entities)
```

## Sortie :

```
[('Elon Musk', 'I-PER'), ('SpaceX', 'I-ORG'), ('Silicon Valley', 'I-LOC')]
```

Copy

Dans cet exemple, Polyglot reconnaît les entités suivantes :

- "Elon Musk" en tant que **personne** (I-PER)
- "SpaceX" en tant qu'**organisation** (I-ORG)
- "Silicon Valley" en tant que **localisation** (I-LOC)

## Conclusion

**Polyglot** est un outil puissant pour effectuer la reconnaissance d'entités nommées en plusieurs langues, en particulier grâce à sa capacité à supporter **plus de 130 langues**. Par rapport à d'autres bibliothèques comme **spaCy** ou **NLTK**, Polyglot est idéal pour les cas où vous devez analyser des textes multilingues sans avoir à spécifier la langue au préalable, car le modèle détecte automatiquement la langue du texte.

->Question : mais aucun de ces exemples utilise plusieurs langues ,pour bien comprendre cette difference entre cet outil polyglot et spacy ou NLTK ??

Excellente remarque ! Tu as tout à fait raison. Pour comparer efficacement Polyglot à spaCy ou NLTK, il faut montrer un exemple multilingue, car c'est là où Polyglot brille vraiment. Voici donc une réponse complète et révisée en Markdown avec :

1. Des exemples concrets en plusieurs langues.
2. Une comparaison directe entre Polyglot, spaCy, et NLTK.
3. Des commentaires et interprétations des résultats.

## Multilingual NER : Polyglot vs spaCy vs NLTK

### Objectif

Montrer la différence entre Polyglot et d'autres bibliothèques comme spaCy et NLTK en termes de support multilingue pour la reconnaissance d'entités nommées (NER).

### Installation des bibliothèques nécessaires

```
pip install polyglot pyicu pyclld2 morfessor
pip install spacy
python -m spacy download en_core_web_sm
```

 Copy

 NLTK ne supporte pas directement le multilingue pour NER, donc on ne l'utilisera que pour l'anglais.

### 1. Exemple Multilingue avec Polyglot

```
from polyglot.text import Text

Texte multilingue : arabe, espagnol, français
text_multi = """
باريس هي عاصمة فرنسا.
Pedro Sánchez es el presidente de España.
Emmanuel Macron est le président de la République française.
"""

Création de l'objet Text
doc = Text(text_multi)
```

 Copy

```
Affichage des entités nommées
for entity in doc.entities:
 print(f"{entity.tag}: {' '.join(entity)}")
```

### ● Sortie attendue :

I-LOC: باريس  
I-LOC: فرنسا  
I-PER: Pedro Sánchez  
I-LOC: España  
I-PER: Emmanuel Macron  
I-LOC: République française

 Copy

 Polyglot détecte automatiquement la langue et identifie correctement les personnes (I-PER) et les lieux (I-LOC) dans 3 langues différentes, sans configuration manuelle. C'est là toute sa force.

## 2. ⚙ spaCy (Anglais uniquement)

```
import spacy

Charger le modèle anglais
nlp = spacy.load("en_core_web_sm")

Texte anglais seulement
text_en = "Pedro Sánchez is the Prime Minister of Spain."

doc = nlp(text_en)

for ent in doc.ents:
 print(ent.label_, ent.text)
```

 Copy

### ● Sortie :

PERSON: Pedro Sánchez  
GPE: Spain

 Copy

 spaCy marche très bien en anglais.

 Mais il ne reconnaîtra pas les entités si tu mets du texte en espagnol, arabe ou français, à moins d'utiliser un modèle spécifique par langue.

### 3. ⚙️ NLTK (Anglais uniquement)

```
import nltk
from nltk import word_tokenize, pos_tag, ne_chunk

nltk.download('maxent_ne_chunker')
nltk.download('words')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

sentence = "Pedro Sánchez is the Prime Minister of Spain."

Tokenisation
tokens = word_tokenize(sentence)
tags = pos_tag(tokens)

Reconnaissance d'entités
tree = ne_chunk(tags)
print(tree)
```

 Copy

#### ● Sortie (structure arborescente) :

```
(S
 (PERSON Pedro/NNP)
 (PERSON Sánchez/NNP)
 is/VBZ
 the/DT
 Prime/NNP
 Minister/NNP
 of/IN
 (GPE Spain/NNP))
```

 Copy

 NLTK reconnaît aussi bien les entités en anglais.

 Mais pas du tout multilingue sans entraîner son propre modèle, ce qui est complexe.

### ⚖️ Comparaison Finale

Bibliothèque	Support Multilingue	Auto-détection de langue	Pré-entraînement NER	Facilité
Polyglot	 130+ langues	 Automatique	 Oui	 
spaCy	 Langue unique par modèle	 Non	 Oui	   
NLTK	 Anglais uniquement	 Non	 Oui (anglais)	 

## Interprétation

- Utilisez **Polyglot** quand :
  - Vous traitez **plusieurs langues** dans un même texte.
  - Vous voulez éviter de **changer de modèle à chaque langue**.
- Utilisez **spaCy** quand :
  - Vous avez des textes **monolingues**.
  - Vous cherchez des performances et une **API facile à utiliser**.
- Utilisez **NLTK** pour des tâches éducatives ou des **textes simples en anglais**.

## Conclusion

**Polyglot** est une excellente bibliothèque **multilingue**, surtout pour des tâches comme **NER sans changement de modèle**. Elle complète bien des outils comme **spaCy** ou **NLTK**, qui sont puissants mais **monolingues par défaut**.

Voici une explication détaillée de l'exercice “French NER with Polyglot I” avec le code, la sortie ligne par ligne, et les interprétations, le tout en Markdown.

# FR French NER with Polyglot I – Explication complète

## Objectif

Utiliser la bibliothèque Polyglot pour :

1. Créer un objet `Text` à partir d'un article en français.
2. Afficher les entités nommées détectées dans le texte.
3. Comprendre leur structure (type Python).

## Rappel : Qu'est-ce qu'une entité nommée ?

Une entité nommée est un mot ou groupe de mots qui correspond à un nom propre : personnes, lieux, organisations, etc.

Exemples :

- *Charles Cuvelliez* → personne
- *Bruxelles* → lieu
- *Deloitte* → organisation

## Le code

```
1. Créer un objet Text à partir du texte de l'article
txt = Text(article)

2. Afficher chaque entité détectée
for ent in txt.entities:
 print(ent)

3. Afficher le type de l'objet ent
print(type(ent))
```

 Copy

## 🖨️ Sortie obtenue

```
['Charles', 'Cuvelliez']
['Charles', 'Cuvelliez']
['Bruxelles']
['l'IA']
['Julien', 'Maldonato']
['Deloitte']
['Ethiquement']
['l'IA']
['.']
<class 'polyglot.text.Chunk'>
```

 Copy

## 🔍 Explication ligne par ligne

```
txt = Text(article)
```

- On crée un objet `Text` à partir de la chaîne `article`.
- Cet objet analyse le texte (tokenisation, détection de la langue, extraction d'entités).
- Polyglot détecte automatiquement que la langue est **le français**.

```
for ent in txt.entities:
```

- On itère sur toutes les entités nommées détectées par Polyglot.
- Chaque `ent` est une **liste de mots** formant une entité (ex: `['Julien', 'Maldonato']`).
- En arrière-plan, ces entités sont des objets de type `Chunk`.

```
print(ent)
```

Voici les entités détectées :

Entité détectée	Interprétation	Type probable
['Charles', 'Cuvelliez']	Nom complet d'une <b>personne</b>	I-PER
['Bruxelles']	Ville / lieu	I-LOC
['l'IA']	Intelligence Artificielle, concept	souvent détecté comme ORG
['Julien', 'Maldonato']	Nom d'une <b>personne</b>	I-PER
['Deloitte']	Entreprise / organisation	I-ORG
['Ethiquement']	Adverbe, probablement faux positif	

Entité détectée	Interprétation	Type probable
[.]	Ponctuation, sans intérêt NER	

**⚠ Note :** Polyglot peut faire quelques erreurs ou extractions douteuses (comme "Ethiquement" ou "."). Cela montre qu'un **post-traitement** peut être nécessaire.

```
print(type(ent))
```

```
<class 'polyglot.text.Chunk'>
```

 Copy

- Chaque entité est un objet de type `Chunk`.
- Ce `Chunk` contient :
  - les mots qui forment l'entité (sous forme de liste),
  - un **label** (`tag`) tel que `I-PER`, `I-LOC`, etc.

Tu pourrais aussi afficher le **type d'entité** comme ceci :

```
for ent in txt.entities:
 print(f"{ent.tag}: {'.join(ent)}")
```

 Copy

## Résumé

Étape	Action	Résultat
Créer <code>Text(article)</code>	Initialise l'analyse linguistique	Détection automatique du français
Itérer sur <code>txt.entities</code>	Extraction des entités nommées	Groupes de mots ( <code>Chunk</code> )
Afficher <code>type(ent)</code>	Comprendre la structure	<code>&lt;class 'polyglot.text.Chunk'&gt;</code>

## À retenir

- Polyglot est très pratique pour les textes multilingues.
- Les entités sont représentées comme des objets `Chunk` contenant :
  - Les mots,
  - Leur catégorie (`I-PER`, `I-ORG`, etc.).
- Il peut être nécessaire de **nettoyer les entités** (par ex. : retirer les ponctuations ou faux positifs).

Voici une explication complète en **Markdown**, ligne par ligne, de l'exercice "Spanish NER with polyglot", qui porte sur l'identification d'entités nommées dans un texte espagnol avec une attention particulière à **Gabriel García Márquez**, alias **Gabo**. ES 

## ES Spanish NER with polyglot — Explication complète

### Objectif

Compter combien d'entités nommées dans le texte se réfèrent à **Gabriel García Márquez**, soit via :

- "Márquez" (nom de famille),
- "Gabo" (surnom).

Ensuite, calculer le **pourcentage** d'entités nommées qui se réfèrent à cette personne.

### Le code

```
Initialiser le compteur
count = 0

Parcourir les entités extraites par polyglot
for ent in txt.entities:
 # Vérifier si l'entité contient 'Márquez' ou 'Gabo'
 if "Márquez" in ent or "Gabo" in ent:
 # Incrémenter le compteur
 count += 1

Afficher le nombre d'entités liées à Márquez / Gabo
print(count)

Calculer le pourcentage d'entités concernées
percentage = count / len(txt.entities)
print(percentage)
```

 Copy

### Explication ligne par ligne

## Ligne 1

```
count = 0
```

 Copy

- Initialise une variable `count` à 0.
- Elle servira à compter le nombre d'entités contenant "Márquez" ou "Gabo".

## Ligne 4

```
for ent in txt.entities:
```

 Copy

- Parcourt chaque entité nommée extraite dans l'objet `txt` (de type `Text`).
- Chaque `ent` est un objet de type `Chunk` représentant une entité nommée (personne, lieu, etc.).

## Ligne 6

```
if "Márquez" in ent or "Gabo" in ent:
```

 Copy

- Vérifie si "Márquez" ou "Gabo" est présent dans l'entité.
- Utilise le mot-clé Python `in`, car `.contains()` n'est pas une méthode valide en Python pour les strings ou listes.

## Pourquoi ça fonctionne :

- `ent` est itérable (liste de mots),
- donc "Gabo" in `ent` vérifie si ce mot est présent dans la liste.

## Ligne 8

```
count += 1
```

 Copy

- Si l'entité contient "Márquez" ou "Gabo", on incrémente `count`.

## Ligne 11

```
print(count)
```

 Copy

- Affiche le **nombre total d'entités** faisant référence à Márquez/Gabo.

## Ligne 14

```
percentage = count / len(txt.entities)
```

 Copy

- Calcule le pourcentage d'entités liées à **Gabriel García Márquez** : [ \text{Pourcentage} = \frac{\text{count}}{\text{nombre total d'entités}} ]

## Ligne 15

```
print(percentage)
```

 Copy

- Affiche le pourcentage trouvé.

## Résultat affiché

```
29
0.29591836734693877
```

 Copy

## Interprétation

- Il y a **29 entités** dans le texte contenant "Márquez" ou "Gabo".
- Cela représente environ **29.6%** de toutes les entités nommées du texte.

## Erreur fréquente expliquée

## Mauvais code

```
if ent.contains("Márquez") or ent.contains("Gabo"):
```

 Copy

## Pourquoi c'est incorrect ?

- `.contains()` n'est pas une méthode Python.
- Les strings et listes utilisent le mot-clé `in` pour vérifier la présence d'un élément.

### Correction :

```
if "Márquez" in ent or "Gabo" in ent:
```

 Copy

### Résumé

Étape	Description
<code>txt.entities</code>	Récupère les entités nommées avec polyglot
<code>ent</code>	Une entité (Chunk), itérable (liste de mots)
Vérification <code>"Márquez" in ent or "Gabo" in ent</code>	Identifie les entités liées à Gabriel G. Márquez
Compteur + pourcentage	Mesure de la présence de cette personne dans l'article