TP — Classes et objets en Python

Token, Sentence, Corpus, méthodes spéciales, héritage

Revekka Kyriakoglou

15 février 2026

But. Construire progressivement un mini-modèle linguistique : `Token` (mot) → `Sentence` (phrase) → `Corpus` (corpus). Vous implémenterez aussi des méthodes spéciales (`__...__`) et une sous-classe `TaggedToken`.

Organisation recommandée (adaptable).

- **Partie A (essentiel, 60–90 min)** : Token + Sentence + Corpus + fréquences
- **Partie B (pythonique, 30–45 min)** : méthodes spéciales
- **Partie C (avance, 20–30 min)** : héritage (TaggedToken)

Règle. Testez *à chaque étape* avec de petits `print`.

Fichier à rendre / à produire : `tp_oop.py`.

0. Mise en place

Créez un fichier `tp_oop.py`.
Lancez :

```
python3 tp_oop.py
```

A. Token (essentiel)

A1. Classe Token (minimum)

Créez une classe `Token` avec :

1. `__init__(self, forme)` : stocke `forme` dans `self.forme`
2. `__str__(self)` : `print(Token("chat"))` affiche `chat`
3. `__repr__(self)` : affichage de debug simple (sans f-string)

Suggestion pour `__repr__` :

```
def __repr__(self):
    return "Token(" + repr(self.forme) + ")"
```

Test à ajouter dans le main :

```
t = Token("Chat")
print(t)        # Chat
print([t])      # [Token('Chat')]
```

A2. Méthodes : pure vs mutante

Ajoutez à `Token` :

1. `normalise(self)` : met `self.forme` en minuscules (mutante)
2. `sans_punct(self)` : renvoie une **nouvelle chaîne** sans `".,;!?"` (pure)
3. `supprime_punct(self)` : remplace `self.forme` par la version sans ponctuation (mutante)

Test :

```
t = Token("Chat,")
print(t.sans_punct(), t.forme)  # Chat  Chat,
t.supprime_punct()
print(t.forme)                  # Chat
t.normalise()
print(t.forme)                  # chat
```

A3 (option). Propriétés `@property`

Ajoutez (optionnel) :

1. `@property longueur` : renvoie `len(self.forme)`
2. `@property est_majuscule` : True si la forme commence par une majuscule
3. `@property est_punct` : True si la forme est dans `".,;!?"`

Test :

```
print(Token("Chat").longueur)        # 4
print(Token("Chat").est_majuscule)   # True
print(Token("!").est_punct)          # True
```

B. Sentence (essentiel)

B1. Classe Sentence (minimum)

Créez une classe `Sentence` qui contient une liste de tokens :

1. `__init__(self, tokens)` : stocke la liste dans `self.tokens`
2. `texte(self)` : reconstruit la phrase (mots séparés par un espace)
3. `__str__(self)` : renvoie `self.texte()`

Test :

```
s = Sentence([Token("Le"), Token("chat"), Token("dort")])
print(s)         # Le chat dort
print(s.tokens)  # liste de Token
```

B2. Fréquences dans une phrase

Ajoutez `frequences(self)` qui renvoie un dictionnaire `mot -> compte` (en minuscules).

Test :

```
s = Sentence([Token("Le"), Token("Chat"), Token("chat")])
print(s.frequences())  # {'le': 1, 'chat': 2}
```

B3. Normaliser une phrase (mutant)

Ajoutez `normalise(self)` qui appelle `normalise()` sur chaque token.

Test :

```
s = Sentence([Token("Le"), Token("Chat"), Token("Dort")])
s.normalise()
print(s)  # Le chat dort
```

C. Corpus (essentiel)

C1. Classe Corpus (minimum)

Créez une classe `Corpus` avec :

1. `__init__(self, sentences)` : stocke la liste dans `self.sentences`
2. `nb_phrases(self)` : renvoie le nombre de phrases
3. `nb_tokens(self)` : renvoie le nombre total de tokens du corpus

Test :

```
s1 = Sentence([Token("Le"), Token("chat")])
s2 = Sentence([Token("Le"), Token("chien"), Token("dort")])
C = Corpus([s1, s2])
print(C.nb_phrases())  # 2
print(C.nb_tokens())   # 5
```

C2. Fréquences globales

Ajoutez `frequences_globales(self)` : dictionnaire `mot -> compte` sur tout le corpus.

Test :

```
print(C.frequences_globales())  # ex: {'le': 2, 'chat': 1, 'chien': 1, 'dort': 1}
```

D. Partie Pythonique : méthodes spéciales (dunder)

D1. Sentence pythonique

Ajoutez à `Sentence` :

1. `__len__` : `len(s)` renvoie le nombre de tokens
2. `__iter__` : `for tok in s: ...` parcourt les tokens
3. `__getitem__` : `s[i]` renvoie le i-ème token
4. `__contains__` : `"chat" in s` (sans tenir compte de la casse)

Test :

```
s = Sentence([Token("Le"), Token("Chat"), Token("dort")])
print(len(s))          # 3
print(s[1])            # Chat (ou Token('Chat') selon __str__/__repr__)
print("chat" in s)     # True
for tok in s:
    print(tok)
```

D2. Corpus pythonique

Ajoutez à `Corpus` :

1. `__len__` : `len(C)` renvoie le nombre de phrases
2. `__iter__` : `for s in C: ...` parcourt les phrases
3. `__getitem__` : `C[i]` renvoie la i-ème phrase
4. `__contains__` : `"chat" in C` True si une phrase contient `"chat"`

Test :

```
print(len(C))          # 2
print(C[0])            # affiche s1
print("chat" in C)     # True
for s in C:
    print(s)
```

E. Partie avancée : héritage (TaggedToken)

E1. Sous-classe TaggedToken

Créez une sous-classe `TaggedToken(Token)` :

1. `__init__(self, forme, pos)` : appelle `super().`$_{init_{(forme)}}$`puisstocke``self.pos`

1. `redéfinissez __repr__ pour afficher forme et pos (sans f-string)`

Test :

```
t = TaggedToken("Chat", "NOUN")
print(t.forme, t.pos)
print([t])
```

E2. Phrase mixte

Construisez une phrase qui mélange `Token` et `TaggedToken` puis normalisez-la :

1. `Sentence([Token("Le"), TaggedToken("Chat","NOUN"), Token("Dort")])`
2. appelez `normalise()` et affichez la phrase

Test :

```
s = Sentence([Token("Le"), TaggedToken("Chat","NOUN"), Token("Dort")])
s.normalise()
print(s)  # le chat dort (selon votre implémentation)
```

Bonus (si temps)

Bonus 1. Ajoutez `__str__` à `Corpus` : une phrase par ligne.
Bonus 2. Ajoutez `Corpus.top_n(n)` : retourne les `n` mots les plus fréquents (sans lambda).
Bonus 3. Ajoutez `Sentence.sans_punct()` : retourne une nouvelle phrase où chaque token est sans ponctuation.