Cours : Classes et objets en Python – 2

Python : objets et classes

présenté par

Revekka Kyriakoglou

le

15 février 2026

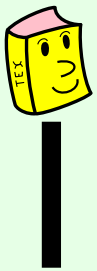
Plan du cours

Rappel : ce qu'on a fait la dernière fois

Points clés

- **Tout est objet** : un objet a un **état** et des **méthodes**.
- **Classe** → **instances** : `t = Token("chat")`.
- `__init__` initialise l'objet ; `self` désigne l'instance courante.
- **Méthodes** : `t.longueur()` ; **attributs** : `t.forme`.
- Méthodes **pures** vs **mutantes** ; attention aux **alias** : `t2 = t1`.

Attributs vs méthodes



Définitions

- **Attribut** : une **donnée** stockée dans l'objet.
- **Méthode** : une **action** (fonction) attachée à l'objet.

```python
class Token:
    def __init__(self, forme):
        self.forme = forme      # attribut

    def longueur(self):
        return len(self.forme)  # méthode

t = Token("chat")
print(t.forme)       # attribut : pas de parenthèses
print(t.longueur())  # méthode  : parenthèses
```

Attribut : on lit / on modifie l'état

```python
t = Token("Chat")
print(t.forme)    # "Chat"

t.forme = "CHAT" # modification directe de l'état
print(t.forme)    # "CHAT"
```

Attribut : on lit / on modifie l'état

```python
t = Token("Chat")
print(t.forme)    # "Chat"

t.forme = "CHAT" # modification directe de l'état
print(t.forme)    # "CHAT"
```

Changer un attribut = changer l'objet (son état).

Méthode : on déclenche une action

```python
class Token:
    def __init__(self, forme):
        self.forme = forme

    def normalise(self):
        self.forme = self.forme.lower()  # méthode mutante

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

Méthode : on déclenche une action

```python
class Token:
    def __init__(self, forme):
        self.forme = forme

    def normalise(self):
        self.forme = self.forme.lower()  # méthode mutante

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

Une méthode peut retourner une valeur et/ou modifier l'objet.

Piège : méthode sans parenthèses

Attention

Sans parenthèses, la méthode n'est pas exécutée : on obtient seulement une référence.

```
t = Token("morphologie")
print(t.longueur)    # référence à la méthode
print(t.longueur())  # exécute la méthode
```

Mini-quiz

Pour chaque expression, dites si c'est un **attribut** ou un **appel de méthode** :

1. `t.forme`
2. `t.longueur`
3. `t.longueur()`
4. `s.upper`
5. `s.upper()`

Mini-quiz

Pour chaque expression, dites si c'est un **attribut** ou un **appel de méthode** :

1. `t.forme`
2. `t.longueur`
3. `t.longueur()`
4. `s.upper`
5. `s.upper()`

Indice : **parenthèses** ⇒ appel (exécution).

Propriétés d'objet (instance) vs propriétés de classe

Deux emplacements, deux comportements

- **Propriété d'objet (instance)** : définie via `self.x` (souvent dans `__init__`)
  $\Rightarrow$ chaque objet a sa propre valeur
- **Propriété de classe** : définie directement dans `class ...:` (hors méthodes)
  $\Rightarrow$ partagée par tous les objets de la classe

Exemple : instance (`self.x`) vs classe

```python
class Token:
    nb_crees = 0          # propriété de classe (partagée)

    def __init__(self, forme):
        self.forme = forme  # propriété d'objet (instance)
        Token.nb_crees += 1

t1 = Token("chat")
t2 = Token("chien")

print(t1.forme, t2.forme)   # chat chien
print(Token.nb_crees)       # 2 (partagé)
```

Attention : ne pas "écraser" une propriété de classe

Piège : Lire `t.nb_crees` marche, mais écrire `t.nb_crees = ...` crée une propriété **d'objet** qui masque la propriété de classe.

```
class Token:
    nb_crees = 0
    def __init__(self, forme):
        self.forme = forme
        Token.nb_crees += 1

t = Token("chat")
print(t.nb_crees, Token.nb_crees)  # 1 1

t.nb_crees = 999
print(t.nb_crees, Token.nb_crees)  # 999 1
```

Pour modifier une propriété de classe : utilisez `Token.nb_crees`, pas `self.nb_crees`.

Exercice : prédire les sorties

Sans exécuter, prédisez l'affichage :

1. `t1 = Token("A")` puis `t2 = Token("B")` puis `print(Token.nb_crees)`
2. ensuite `t1.nb_crees = 100` puis `print(t1.nb_crees, t2.nb_crees, Token.nb_crees)`

Exercice : compteur d'objets

Dans la classe `Token` :

1. Ajoutez un attribut de classe `nb_crees`.
2. Incrémentez-le dans `__init__`.
3. Créez 3 tokens et affichez `Token.nb_crees`.

Exercice : compteur d'objets

Dans la classe `Token` :

1. Ajoutez un attribut de classe `nb_crees`.
2. Incrémentez-le dans `__init__`.
3. Créez 3 tokens et affichez `Token.nb_crees`.

Règle : on modifie un attribut de classe via `Token.nb_crees`, pas via `self.nb_crees`.

Correction

```python
class Token:
    nb_crees = 0

    def __init__(self, forme):
        self.forme = forme
        Token.nb_crees += 1

t1 = Token("chat")
t2 = Token("chien")
t3 = Token("dort")

print(Token.nb_crees)  # 3
```

Attribut calculé : `@property`

Idée

Un **attribut calculé** se lit comme un attribut, mais exécute un calcul.

- lecture : `t.longueur`
- pas de parenthèses
- (en général) pas d'effet de bord

```
class Token:
    def __init__(self, forme):
        self.forme = forme
    @property
    def longueur(self):
        return len(self.forme)

t = Token("morphologie")
print(t.longueur)  # 10
```

`@property` : faut-il l'écrire dans le code ?

Réponse

Oui, si vous voulez accéder à la valeur **sans parenthèses** : `t.longueur`. Sans `@property`, il faut une méthode : `t.longueur()`.

```
class Token:
    def __init__(self, forme): self.forme = forme
    @property
    def longueur(self): return len(self.forme)

t = Token("chat")
print(t.longueur)
```

Quand utiliser `@property` ?

Règle pratique

- **Oui** : valeur dérivée simple, rapide, **sans modifier** l'objet.
- **Non** : calcul lourd, accès réseau/fichier, ou méthode qui modifie l'état.

Exemples : `longueur`, `est_majuscule`, `est_punct`.

Exercice : une propriété utile

Dans `Token`, ajoutez :

1. une propriété `est_majuscule` (True si la forme commence par une majuscule)
2. une propriété `est_punct` (True si la forme est dans `".,;!?"`)

Testez sur `"Chat"`, `"dort"`, `"!"`.

Composition : une structure en 3 niveaux

Idée

On modélise un mini-texte en linguistique :

- **Token** : un mot (ex : `"chat"`)
- **Sentence** : une phrase = **liste de Token**
- **Corpus** : un corpus = **liste de Sentence**

Étape 1 : construire une phrase (sans classe)

Avant la classe

On crée des objets `Token`, puis on les met dans une liste.

```
t1 = Token("Le")
t2 = Token("chat")
t3 = Token("dort")

tokens = [t1, t2, t3]
print(tokens)
```

Étape 1 : construire une phrase (sans classe)

Avant la classe

On crée des objets `Token`, puis on les met dans une liste.

```
t1 = Token("Le")
t2 = Token("chat")
t3 = Token("dort")

tokens = [t1, t2, t3]
print(tokens)
```

La liste `tokens` contient des objets `Token`, pas des chaînes.

Étape 2 : classe `Sentence` (minimum)

On encapsule la liste

Une phrase est un objet qui stocke une liste de tokens.

```
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

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

Étape 3 : méthode `texte()` (reconstruire la phrase)

```
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

    def texte(self):
        mots = []
        for tok in self.tokens:
            mots.append(tok.forme)
        return " ".join(mots)

s = Sentence([Token("Le"), Token("chat"), Token("dort")])
print(s.texte())
```

Exercice (Sentence) : méthode `texte()`

Complétez `Sentence.texte()` :

1. créer une liste vide `mots`
2. parcourir `self.tokens` et ajouter `tok.forme`
3. retourner `" ".join(mots)`

Test : `Sentence([Token("Le"), Token("chat")]).texte()` affiche `Le chat`.

Correction (Sentence) : méthode `texte()`

```python
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

    def texte(self):
        mots = []
        for tok in self.tokens:
            mots.append(tok.forme)
        return " ".join(mots)

# Test
print(Sentence([Token("Le"), Token("chat")]).texte())  # Le chat
```

Étape 4 : fréquences dans une phrase

```
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

    def frequences(self):
        freq = {}
        for tok in self.tokens:
            mot = tok.forme.lower()
            if mot in freq:
                freq[mot] += 1
            else:
                freq[mot] = 1
        return freq

s = Sentence([Token("Le"), Token("Chat"), Token("chat")])
print(s.frequences())
```

Exercice (Sentence) : `frequences()`

Écrivez `Sentence.frequences()` sans `get` :

1. `freq = {}`
2. pour chaque token : `mot = tok.forme.lower()`
3. si `mot` est dans `freq` : incrémenter, sinon initialiser à 1

Correction (Sentence) : frequences()

```python
class Sentence:
    def __init__(self, tokens):
        self.tokens = tokens

    def frequences(self):
        freq = {}
        for tok in self.tokens:
            mot = tok.forme.lower()
            if mot in freq:
                freq[mot] += 1
            else:
                freq[mot] = 1
        return freq

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

Étape 5 : classe Corpus (minimum)

```
class Corpus:
    def __init__(self, sentences):
        self.sentences = sentences

C = Corpus([
    Sentence([Token("Le"), Token("chat")]),
    Sentence([Token("Le"), Token("chien"), Token("dort")])
])
print(len(C.sentences))  # 2
```

Corpus : nombre de phrases et nombre de tokens

```
class Corpus:
    def __init__(self, sentences):
        self.sentences = sentences

    def nb_phrases(self):
        return len(self.sentences)

    def nb_tokens(self):
        total = 0
        for s in self.sentences:
            total += len(s.tokens)
        return total
```

Corpus : nombre de phrases et nombre de tokens

```
class Corpus:
    def __init__(self, sentences):
        self.sentences = sentences

    def nb_phrases(self):
        return len(self.sentences)

    def nb_tokens(self):
        total = 0
        for s in self.sentences:
            total += len(s.tokens)
        return total
```

On compte les tokens en additionnant la taille de chaque phrase.

Exercice final : mini-modèle linguistique

1. Construisez `s1` et `s2` (2–5 tokens chacune).
2. Construisez `C = Corpus([s1, s2])`.
3. Vérifiez :
   - `s1.texte()` et `s1.frequences()`
   - `C.nb_phrases()` et `C.nb_tokens()`

Exercice final : mini-modèle linguistique

1. Construisez `s1` et `s2` (2–5 tokens chacune).
2. Construisez `C = Corpus([s1, s2])`.
3. Vérifiez :
   - `s1.texte()` et `s1.frequences()`
   - `C.nb_phrases()` et `C.nb_tokens()`

Tester **Sentence** d'abord, puis seulement ensuite **Corpus**.

Transition : rendre nos objets « comme Python »

Idée

On veut que nos objets fonctionnent naturellement avec Python :

- afficher (`print(...)`)
- taille (`len(...)`)
- itération (`for ... in ...`)
- indexation (`obj[i]`)
- appartenance (`x in obj`)

Transition : rendre nos objets « comme Python »

Idée

On veut que nos objets fonctionnent naturellement avec Python :

- afficher (`print(...)`)
- taille (`len(...)`)
- itération (`for ... in ...`)
- indexation (`obj[i]`)
- appartenance (`x in obj`)

Ces comportements déclenchent des méthodes spéciales : `__...__`.

Méthodes spéciales (dunder)

Idée

Certaines méthodes permettent d'utiliser des fonctions/opérateurs Python :

- `print(obj)` → `__str__` (ou `__repr__`)
- `len(obj)` → `__len__`
- `for x in obj` → `__iter__`
- `obj[i]` → `__getitem__`
- `x in obj` → `__contains__`

`__str__` : que fait `print(obj)` ?

Idée

Quand on écrit `print(obj)`, Python cherche une méthode `__str__`.

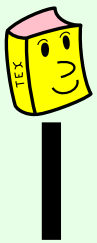
```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def __str__(self):
        return self.forme

t = Token("chat")
print(t)
```

Exercice : `__str__`

Ajoutez `__str__` à `Token` pour que :

- `print(Token("Chat"))` affiche `Chat`
- `print(Token("dort"))` affiche `dort`

__repr__ : affichage pour le debug



Idée

__repr__ est utilisé quand Python affiche des objets dans des listes.

```
class Token:
    def __init__(self, forme):
        self.forme = forme

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

print([Token("Le"), Token("chat")])
```

Exercice : `__repr__`

Ajoutez `__repr__` à `Token` pour que :

- `print([Token("Le"), Token("chat")])` affiche une liste lisible
- format attendu : `Token('chat')`

`__len__` : que fait `len(obj)` ?

Idée

Quand on écrit `len(obj)`, Python cherche `__len__`.

```
class Token:
    def __init__(self, forme):
        self.forme = forme

    def __len__(self):
        return len(self.forme)

print(len(Token("morphologie")))
```

Exercice : `__len__`

Ajoutez `__len__` à `Token` pour que :

- `len(Token("chat")) == 4`
- `len(Token("morphologie")) == 10`

`__iter__` : que fait `for x in obj` ?

Idée

Dans `Sentence`, `__iter__` doit renvoyer un itérateur sur `self.token`

```
class Sentence:
    def __iter__(self):
        return iter(self.tokens)
```

Exercice : `__iter__`

Dans `Sentence`, ajoutez `__iter__` pour que :

- `for tok in s: ...` fonctionne
- `list(s)` renvoie la liste des tokens

`__getitem__` : que fait `obj[i]` ?

Idée

Dans `Sentence`, `__getitem__` renvoie un token par index.

```
class Sentence:
    def __getitem__(self, i):
        return self.tokens[i]
```

Exercice : `__getitem__`

Dans `Sentence`, ajoutez `__getitem__` pour que :

- `s[0]` renvoie le premier token
- `s[-1]` renvoie le dernier token

Testez sur une phrase de 3 tokens.

`__contains__` : que fait `x in obj` ?

Idée

Dans `Sentence`, `__contains__` teste si un mot est présent (sans casse).

```python
class Sentence:
    def __contains__(self, forme):
        cible = forme.lower()
        for tok in self.tokens:
            if tok.forme.lower() == cible:
                return True
        return False
```

Exercice : `__contains__`

Dans `Sentence`, ajoutez `__contains__` pour que :

- `"chat" in s` renvoie True si un token a la forme `"chat"` (sans tenir compte de la casse)
- `"chien" in s` renvoie False si absent

Checkpoint A : `Token` complet

Objectif : avoir une classe `Token` utilisable dans la suite.

1. Ajoutez `normalise(self)` (met `self.forme` en minuscules).
2. Ajoutez `@property est_majuscule` (True si première lettre majuscule).
3. Ajoutez `@property est_punct` (True si forme dans `".,;!?"`).
4. (option) Ajoutez `@property longueur` (retourne `len(self.forme)`).

Test :

- `Token("Chat").est_majuscule`
  `Token("!").est_punct`
- `t = Token("Chat"); t.normalise();`
  `print(t.forme)`

Checkpoint B : mini-modèle

Objectif : tout faire fonctionner ensemble.

1. `Sentence.texte()` : reconstruit la phrase.
2. `Sentence.frequences()` : dictionnaire mot→compte.
3. `Corpus.nb_tokens()` : nombre total de tokens.

Test minimal :

- `s1 = Sentence([Token("Le"), Token("Chat")])`
- `s2 = Sentence([Token("Le"), Token("chien"), Token("dort")])`
- `C = Corpus([s1, s2])`
- vérifier `s1.texte()`, `s1.frequences()`, `C.nb_tokens()`

Checkpoint C : objets « Pythonic »

Objectif : utiliser les syntaxes Python naturelles.

1. Dans `Sentence` : `__len__`, `__iter__`, `__getitem__`, `__contains__`.
2. (option) Dans `Token` : `__str__`, `__repr__`, `__len__`.

Test :

- `len(s)`, `s[0]`, `"chat" in s`, `for tok in s: ...`
- `print([Token("Le"), Token("chat")])`

Héritage : quand et pourquoi ?

Idée

Héritage = relation « est-un » :

- un `TaggedToken` **est un** `Token`
- mais il a **plus d'information** (ex : `pos`)

Héritage : quand et pourquoi ?

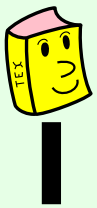
Idée

Héritage = relation « est-un » :

- un `TaggedToken` **est un** `Token`
- mais il a **plus d'information** (ex : `pos`)

À comparer avec la composition : `Sentence` **a des** `Token`.

Créer une sous-classe : TaggedToken(Token)



Point clé

`super()`.$_{init(forme)}$ *initialiselapartiehrite*(`Token`).

```
class TaggedToken(Token):
    def __init__(self, forme, pos):
        super().__init__(forme)  # initialise Token
        self.pos = pos           # nouvel attribut

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

Override : redéfinir __repr__ (simple)

```
class TaggedToken(Token):
    def __init__(self, forme, pos):
        super().__init__(forme)
        self.pos = pos

    def __repr__(self):
        return "TaggedToken(" + repr(self.forme) + ", " +
        ↪  repr(self.pos) + ")"

print([TaggedToken("chat", "NOUN")])
```

Polymorphisme : même méthode, objets différents

Idée

Si `TaggedToken` hérite de `Token`, on peut utiliser les mêmes méthodes (ex : `normalise`).

```
tokens = [Token("Le"), TaggedToken("Chat", "NOUN"),
↪  Token("Dort")]
for tok in tokens:
    tok.normalise()  # marche pour Token et TaggedToken
print([tok.forme for tok in tokens])  # ['le', 'chat', 'dort']
```

Exercice : TaggedToken et phrase mixte

Créez une classe `TaggedToken(Token)` avec `pos`.

1. Utilisez `super().__init__(forme)`.
1. Redéfinissez `__repr__` pour afficher forme et pos.
2. Construisez une phrase mixte : `Sentence([Token("Le"), TaggedToken("Chat","NOUN"), Token("dort")])`.
3. Normalisez la phrase et affichez les formes.